Erweiterung von SCCharts um Datenfluss

Axel Umland

Diplomarbeit
eingereicht am 25. März 2015

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe für Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden

Betreut durch:
Dipl.-Inf. Steven Smyth,
Dipl.-Inf. Ass. iur. Insa Fuhrmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Der Entwurf eingebetteter, reaktiver Systeme erfolgt häufig modellbasiert. Viele Sprachen bieten dabei entweder nur das datenflussbasierte oder kontrollflussbasierte Modellierungskonzept an. Kontrollflussdiagramme haben ihren Schwerpunkt in der Visualisierung von Verhaltensbeschreibungen von Systemen. Der Schwerpunkt von Datenflussdiagrammen liegt bei der Darstellung von Berechnungs- und Kommunikationsfluss zwischen einzelnen Elementen in einem Modell. Eine Entwicklungsumgebung die beide Konzepte zulässt, erweitert die Benutzerfreundlichkeit und Lesbarkeit von Modellen. So können für jeden Teilaspekt eines Modells die Vorteile der entsprechenden Darstellung genutzt werden.

SCCharts sind eine visuelle Modellierungssprache zur Beschreibung von Kontrollflussdiagrammen, basierend auf dem sequentiell konstruktiven Berechnungsmodell. Diese Diplomarbeit stellt ein Konzept zur Erweiterung von SCCharts um Datenfluss und dessen Implementierung in der KIELER Umgebung vor. Dies beinhaltet die textuelle Modellierung von Datenfluss, sowie die dazugehörige Visualisierung als neuer Bestandteil der SCCharts. Des Weiteren wird eine Modelltransformation implementiert, welche den Datenflussanteil in klassische SCCharts übersetzt. Der Funktionsumfang der Implementierung wird anhand diverser Beispielmodelle vorgestellt und ausgewertet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

DSL Domain Specific Language

EMF Eclipse Modeling Framework

EPL Eclipse Public License

ID Identifier

IDE Integrated Development Environment

KExpression KIELER Expression

KiCo KIELER Compiler

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

KIML KIELER Infrastructure for Meta Layout

KLay KIELER Layout Algorithms

KLighD KIELER Lightweight Diagrams

M2M Model-to-Model

MoC Model of Computation

RCP Rich Client Platform

SSM Safe State Machine

SCADE Safety-Critical Application Development Environment

SMoC Synchronous MoC

SCMoC Sequentially Constructive MoC

SCChart Sequentially Contructive Chart

SCG SC Graph

SLIC Single-Pass Language-Driven Incremental Compilation

XMI XML Metadata Interchange

XML Extensible Markup Language

Kapitel 1

Einleitung

Die Verwendung eingebetteter und reaktiver Systeme ist fast überall im Alltag zu finden. In vielen Bereichen, wie zum Beispiel in der Automobil- oder Eisenbahnindustrie, haben diese Systeme sicherheitskritische Anforderungen. Ein Airbag im Auto hat besispielsweise solche kritischen Anforderungen. So würde ein Fehlverhalten der Ansteuerung, eine verzögerte Ausführung oder eine fehlerhafte Überwachung des Airbag-Systems die Sicherheit von Personen gefährden.

Ein reaktives System erhält ständig Eingabedaten von seiner Umgebung und berechnet daraus die Ausgaben, welche es an die Umgebung zurück gibt. Abbildung 1.1 zeigt den schematischen Aufbau eines reaktiven Systems, eingebettet in einer Umgebung.

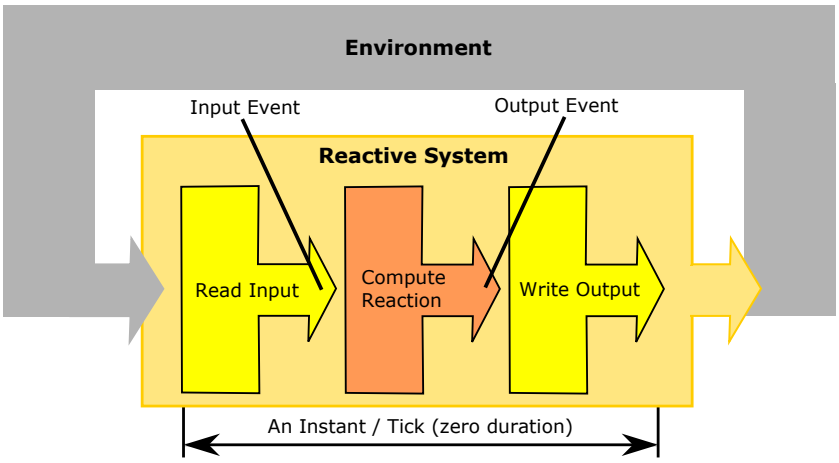


Abbildung 1.1. Schema eines eingebetteten, reaktiven Systems [MvHH13]

Für die Entwicklung sicherheitskritischer Softwaresysteme bieten sich häufig synchrone Sprachen an. Aufgrund ihrer Determiniertheit und Vorhersagbarkeit haben synchrone Sprachen gegenüber anderen Programmiersprachen einen klaren Vorteil. Das klassische synchrone Berechnungsmodell (*synchronous Model of Computation* (SMoC)) unterteilt die Zeit in diskrete Einheiten (Ticks). Dabei wird angenommen, dass jegliche Berechnungen, die ein reaktives System ausführt, keine Zeit verbrauchen [BB91]. In Abbildung 1.1 ist so ein Tick exemplarisch dargestellt zwischen den Komponenten „Read Input“ und „Write Output“.

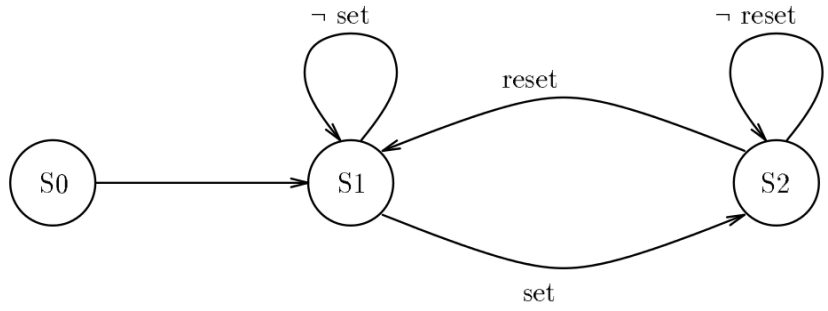
Die graphische Repräsentation von Funktionen, Programmen, Modellen und Systemen ist weit verbreitet. Diagramme von Modellen können erheblich zum besseren Verständnis von komplexen Systemen beitragen. Statecharts sind ein von David Harel eingeführter Formalismus zur Beschreibung komplexer, unter anderem ereignisgesteuerter, Echtzeitsysteme und stellen eine Erweiterung klassischer Zustandsübergangsdiagramme dar [Har87]. Eine Übertragung dieser Statechart-Notation auf die synchrone textuelle Sprache Esterel [BG92] wurde 1996 von Charles André mit den SyncCharts vorgestellt. Jedes SyncChart kann in ein äquivalentes Esterel-Programm übersetzt werden, wodurch diese Diagramme als eine graphische Repräsentation von Esterel angesehen werden können [And96b]. Sequentially Contructive Charts (SCCharts) sind eine visuelle Modellierungssprache zur Entwicklung sicherheitskritischer, reaktiver Systeme [vHDM+14]. Neben der Statechart-Notation verwenden sie statt des klassischen synchronen MoC das sequentiell konstruktive Berechnungsmodell (Sequentially Constructive Model of Computation (SCMoC)). Damit sind SCCharts weniger restriktiv als SyncCharts, die das klassische synchrone MoC verwenden, aber alle validen SyncCharts sind auch valide SCCharts [vHDM+14].

1.1. Motivation und Problemstellung

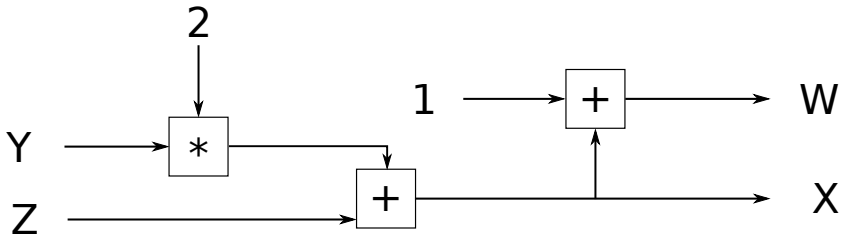
Viele synchrone Sprachen bieten entweder nur die Möglichkeit der Modellierung von Kontrollfluss (beispielsweise Esterel [BG92]) oder Datenfluss (zum Beispiel Lustre [HCRP91] oder Signal [GGBM91]). Die Beschreibung kontrollflussdominierter Anwendungen erfolgt häufig unter Verwendung hierarchischer, synchroner Zustandsautomaten, beispielsweise der Statechart-Notation entsprechend. Bei Anwendungen, in denen die Berechnung von Werten und der damit verbundene Kommunikationsfluss zwischen einzelnen Komponenten im Vordergrund steht, bieten sich eher deklarative Datenflusssprachen an. Die graphische Repräsentation erfolgt dabei häufig in Form von Blockdiagrammen. Synchrone Datenflusssprachen bestimmen durch (interne) Clockfunktionen wann die Berechnung eines Wertes ausgeführt werden soll.

Abbildung 1.2a zeigt ein einfaches Beispiel eines Kontrollflusses, in dem nur mit dem Signal `set` aus dem Zustand `S1` in den Zustand `S2` gewechselt wird und nur mit dem Signal `reset` wieder zurück. In Abbildung 1.2b ist die Datenflussbeschreibung von zwei Gleichungen, einmal `X = 2 * Y + Z` und `W = X + 1` dargestellt.

Nicht jedes Anwendungsbeispiel lässt sich eindeutig einer dieser beiden Kategorien zuordnen. Mit einer Kombination beider Modellierungskonzepte wird dem Benutzer die Handhabung dahingehend vereinfacht, dass Kontrollflussstrukturen in Datenflussdiagrammen einfacher umgesetzt und dargestellt werden können. Das gilt auch umgekert für die Darstellung von Datenfluss als



(a) Beispiel eines Kontrollflussdiagramms [HCRP91]



(b) Beispiel zur Darstellung von Datenfluss, nach [HCRP91]

Abbildung 1.2. Beispiele der Modellierungskonzepte

Blockdiagramme innerhalb einer Kontrollflussumgebung. Die Entscheidung welche Modellierungssprache für welchen Teil einer Anwendung sinnvoller ist, würde entfallen.

Ziel meiner Diplomarbeit ist es, SCCharts um die Möglichkeit der Datenflussmodellierung zu erweitern. Da SCCharts weniger Limitierungen als SyncCharts haben, bieten sie inhaltlich bereits jetzt dem Benutzer mehr Freiheiten und Möglichkeiten in der Modellierung. Zur Verdeutlichung wann eine Datenflussnotation mit entsprechender Visualisierung von Vorteil ist, sei das Beispiel einer Transition genannt. Beim Zustandsübergang kann einer zu emittierenden Variablen direkt ein Wert zugewiesen werden oder durch eine komplexe Berechnung. Das Auftreten so einer komplexen Berechnung kann das Verständnis der Transition beeinträchtigen. Für die bessere Lesbarkeit der Modelle ist eine Darstellung solcher komplexer Berechnungen als datenflussorientiertes Blockdiagramm von Vorteil. Insbesondere der Kommunikationsfluss zwischen einzelnen Modellelementen lässt sich in Form eines Datenflussblockdiagramms sehr gut kenntlich machen.

Eine Orientierung, wie eine kombinierte Modellierung beider Konzepte möglich ist, bietet die von Esterel Technologies[1] entwickelte Safety-Critical Application Development Environment (SCADE) [Ber07], welche eine Erweiterung der

[1] http://www.esterel-technologies.com/

synchronen Datenflusssprache Lustre ist. Mit SCADE lassen sich modellbasierte eingebettete Softwaresysteme entwickeln und validieren, welche beliebig verschachtelten Daten- und Kontrollfluss enthalten können. Die zugrundeliegende deklarative synchrone Datenflusssprache Lustre wurde dabei um die Modellierung von Kontrollfluss durch Safe State Machines (SSMs) erweitert. In Abbildung 1.3 ist ein Beispiel der kombinierten Modellierung von Daten- und Kontrollfluss in SCADE dargestellt. Der Kontrollfluss wird durch das Zustandsdiagramm „Regulate“ mit den beiden Zuständen „RegulOn“ und „StandBy“ dargestellt. Der Datenfluss wird in Form von Blockdiagrammen innerhalb dieser Zustände angezeigt. Im Startzustand „RegulOn“ beinhaltet das Datenflussblockdiagramm die Eingangsdaten cruise_speed und Speed, die von der Komponente Regulator verarbeitet werden und den Ausgangswert ThrottleCmd berechnen. Im Zustand „Standby“ wird dieser Ausgangswert anhand der Komponente MngThrottle und dem Eingangswert Accel berechnet. Der Zustandswechsel wird durch die beiden Werte accelerating und between gesteuert.

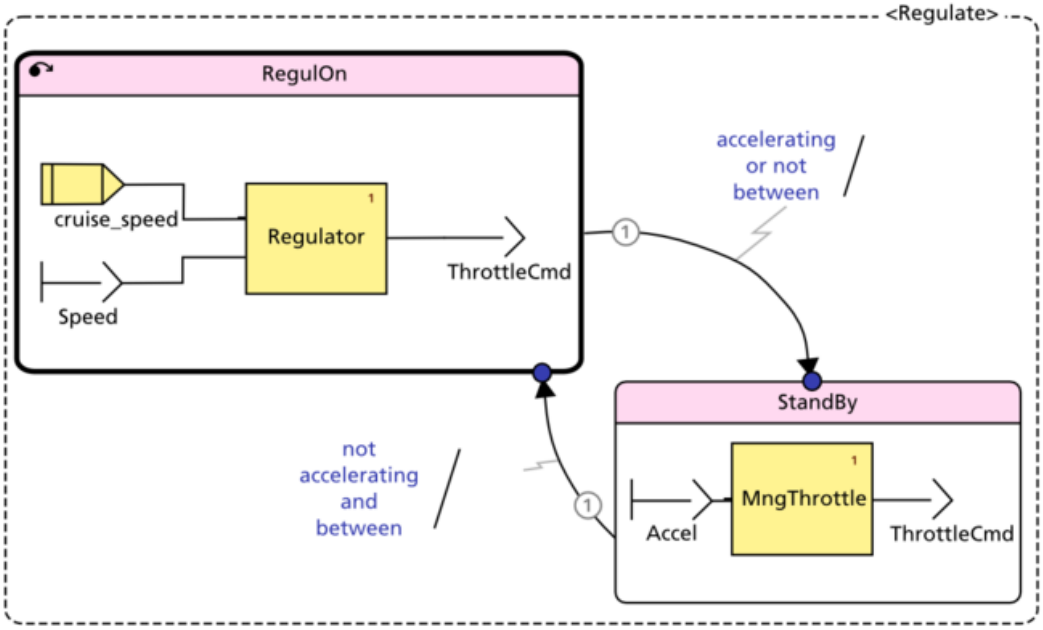


Abbildung 1.3. Beispiel zur Kombination von Daten- und Kontrollfluss in einem SCADE-Modell (Wikipedia)

1.2. Beitrag

Diese Diplomarbeit erweitert die Modellierungssprache der SCCharts innerhalb des KIELER Projekts um Datenfluss. Mit dem Ergebnis meiner Arbeit ist es möglich komplexe Datenflusskomponenten zu definieren, zu verwenden und zu veranschaulichen ohne diese vorher manuell in ein äquivalentes SCChart zu übersetzen. Die Übersetzung erfolgt automatisch durch eine Modelltransformation, so dass Modelle mit beliebig verschachtelter Kombination von Daten- und Kontrollfluss weiterhin valide SCCharts bleiben. Die Erweiterung erfolgt nicht auf direkter sprachlicher Ebene. Eine Einbindung neuer Operatoren für SCCharts, oder eine Anpassung der KExpressions erfolgt nicht.

1.3. Gliederung

Diese Diplomarbeit beschreibt die Erweiterung von SCCharts um Datenfluss und ist wie folgt unterteilt.

Zunächst wird in Kapitel 2 ein Überblick über verwandte Arbeiten gegeben. In Kapitel 3 schließt sich eine Vorstellung der verwendeten Technologien an. Kapitel 4 beschreibt das Konzept zur Erweiterung von SCCharts um Datenfluss. Dies beinhaltet die Vorstellung der Notation von Datenfluss und die Erweiterungen am Metamodell und der Grammatik von SCCharts. Des Weiteren wird die Visualisierung mittels KLighD, sowie die Anpassung der benötigten Modelltransformation beschrieben. Danach folgt die Beschreibung der Implementierung in Kapitel 5. Eine Auswertung der Implementierung erfolgt in Kapitel 6 und gibt einen Überblick des erreichten Funktionsumfangs. Abschließend wird die Diplomarbeit in Kapitel 7 zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben.

Kapitel 2

Verwandte Arbeiten

Synchrone Sprachen wurden mit dem Ziel entwickelt, dass sie nebenläufiges, deterministisches Verhalten korrekt beschreiben [BCE+03]. Drei der bekanntesten Vertreter dieser Programmiersprachen sind Esterel [BdS91], Lustre [HCRP91] und Signal [GGBM91]. Die Entwicklung reaktiver, sicherheitskritischer Systeme profitiert besonders von den Eigenschaften synchroner Sprachen. Das klassische synchrone Berechnungsmodell (*synchronous Model of Computation* (SMoC)) nimmt an, dass die Zeit in diskrete Einheiten aufgeteilt ist und die Berechnungen jeweils für eine solche Einheit, den sogenannten (Makro-)Tick stattfinden [BB91]. Diese Zeiteinheiten lassen sich wiederum unterteilen in eine endliche Anzahl von Mikroticks, welche die einzelnen Berechnungsschritte darstellen.

Die synchrone Hypothese Die Annahme eines perfekten synchronen Systems besagt, dass dieses System keine Zeit zwischen zwei Ticks verbraucht um alle Berechnungen durchzuführen. Daraus ergibt sich, dass die Ausgangsdaten bereits zu dem Zeitpunkt feststehen, wenn die Eingangsdaten gelesen werden. In der Praxis bedeutet diese Annahme, dass wenn Eingangsdaten auftreten, beziehungsweise sich verändern, das System schnell genug reagiert um diese Daten zu verarbeiten bevor die nächsten Eingangsdaten auftreten [Gam10].

Sequentielle Konstruktivität Die deterministische Nebenläufigkeit synchroner Sprachen wird durch hohe Restriktivität erreicht. Programme, welche die entsprechenden Bedingungen erfüllen werden valide, oder auch *konstruktiv* genannt. Von Hanxleden et al. [vHMA+13] haben mit dem *sequentiell konstruktiven* Berechnungsmodell (Sequentially Constructive Model of Computation (SCMoC)) eine konservative Erweiterung des klassischen synchronen MoC vorgestellt. Es hebt einige Einschränkungen des synchronen MoC auf, so dass zum Beispiel gemeinsam genutzte Variablen in einem Tick beliebig oft gelesen und geschrieben werden können, sofern sich ein zulässiges sequentiell konstruktives Scheduling finden lässt [vHMA+13].

Datenfluss versus Kontrollfluss Eine wichtige Anforderung an eine synchrone Sprache ist die Unterstützung von Nebenläufigkeit, wobei diese benutzerfreundlich programmierbar sein soll [BCE+03]. Wie bereits in Unterkapitel 1.1 erläutert,

eignet sich dazu abhängig von der Zielanwendung entweder eine Notation als Datenflussblockdiagramm oder als hierarchischer Automat. Die Verwendung einer Automatennotation eignet sich besonders für kontrollflussdominierte Anwendungen. Für Anwendungen, in denen es eher darum geht welche Daten in welcher Weise verarbeitet werden, eignet sich hingegen eine Datenflussblocknotation. Mit dieser Notation lässt sich unter anderem auch der Kommunikationsfluss zwischen einzelnen Komponenten einfach und leicht verständlich darstellen.

Mit der Zeit entwickelte sich der Anspruch auf die kombinierte Verwendung beider Notationsvarianten [BCE+03]. Das folgende Unterkapitel erläutert vorhandene Varianten zur Erweiterung einer Datenflussnotation um Kontrollfluss. Danach werden im Unterkapitel 2.2 Hybridansätze zur Kombination verschiedener Notationen und Berechnungsmodelle kurz vorgestellt.

2.1. Erweiterung von Datenfluss um Kontrollfluss

Die Entwicklung berechnungsintensiver, reaktiver Systeme erfolgt häufig mittels deklarativer synchroner Sprachen. Ein Beispiel diser Sprachfamilie ist Lustre. Diese Sprache wird hauptsächlich zur Programmierung von eingebetteten Echtzeitsysteme verwendet, insbesondere zur Entwicklung automatischer Kontrollsysteme und Signalverarbeitungssysteme [CPHP87].

Ein Lustre-Programm arbeitet mit Sequenzen (sogenannten *flows*) von Werten. Jede Variable x und jeder Ausdruck e wird als eine unendliche Sequenz $(x_0, x_1, \ldots, x_n, \ldots)$ interpretiert. Dabei ist x_n der Wert der Variablen x zum Zeitpunkt n des Systems. Eine Gleichung der Form $x = e$ wird demnach wie folgt interpretiert $\forall n : x_n = e_n$ [CPHP87]. Die Zeit wird wie in synchronen Sprachen üblich in diskrete Einheiten unterteilt und ebenfalls als eine Sequenz von Werten interpretiert. So eine Sequenz der Zeit wird *clock* genannt. Unter Verwendung von Sequenzen mit booleschen Werten lassen sich langsamere und schnellere clocks definieren. Damit lassen sich dann Komponenten im Datenfluss ansteuern, so dass deren Berechnungen nur ausgeführt werden, wenn ihre clock den Wert *true* annimmt [HCRP91].

Bei größeren Anwendungen wird es zunehmend unübersichtlicher, wann welche Komponenten angesteuert und ausgeführt werden. Maraninchi und Rémond haben eine Variante zur Erweiterung von synchronen deklarativen Datenflusssprachen um verschiedene Modi (Mode-Automata) vorgestellt [MR98, MR03]. Mit diesen Automaten zur Emulierung verschiedener Modi, in denen sich ein Programm befinden kann, ist es einfacher Kontrollfluss in diesen Programmen zu modellieren. Dieser Ansatz umfasst auch Konstrukte zur Kombination solcher Automaten, wie zum Beispiel Parallelität und Hierarchie.

Eine Erweiterung auf direkter sprachlicher Ebene zur Einbindung von Kontrollfluss ist die Verwendung von Zustandsautomaten. Colaço et al. [CP03, CPP05]

zeigen, wie man die booleschen clocks nutzen kann, um die imperativen Konstrukte dieser Zustandsautomaten in wohldefinierte Programme der Basissprache zu übersetzen.

Eine weitere Variante stellt die Safety-Critical Application Development Environment (SCADE) dar. SCADE entwickelte sich aus Saga, welches ein Werkzeug zur graphischen Repräsentation von Lustre-Programmen ist [BP88]. Der synchrone Datenfluss wird hier ebenfalls als Blockdiagramm dargestellt. Für die Entwicklung von Anwendungen mit der kombinierten Verwendung von Daten- und Kontrollfluss, wurde SCADE um Safe State Machines (SSMs) erweitert [Ber07]. Diese Erweiterung um hierarchische SSMs basiert auf der synchronen Programmiersprache Esterel [BG92] und dem synchronen Statechart-Modell der SyncCharts [And96a].

2.2. Hybridansätze

Ein Beispiel zur Kombinierung verschiedener Ansätze ist das Ptolemy II[1] Projekt. Ptolemy verfolgt den Ansatz der aktororientierten Entwicklung. Aktoren sind einzelne Softwarekomponenten, die nebenläufig agieren und auch hierarchisch aufgebaut sind. Das besondere Merkmal von Ptolemy ist, dass das Verhalten der einzelnen Softwarekomponenten von einem sogenannten *Director* beschrieben wird. Diesen Directors können dabei unterschiedliche Berechnungsmodelle zu Grunde liegen. Dadurch entsteht eine hierarchische Heterogenität, die die unterschiedlichen Anforderungen verschiedener Berechnungsmodelle der Teilkomponenten eines Modells beachtet [EJL+03].

In Kombination mit endlichen Zustandsautomaten lassen sich modale Modelle beschreiben [Lee09]. Diese Modelle erweitern die Zustandsautomaten derart, dass mehrere Verhaltensweisen statt nur einer in einem Zustand beschrieben werden können. Eine Zustandsmaschine kontrolliert dabei zu jedem Zeitpunkt welches Verhalten ausgewählt werden soll.

Eine Anwendung von Hybridsystemen mit Zustandsautomaten in Kombination mit diskreten und kontinuierlichen Ereignissen wird in [LZ05] am Beispiel der Sprache HyVisual [BCL+05] beschrieben.

[1] http://ptolemy.eecs.berkeley.edu/

Kapitel 3

Verwendete Technologien

Die Erweiterung von SCCharts um Datenfluss erfolgt durch die Anpassung bestehender Eclipse Plugins innerhalb der KIELER Umgebung. Zum besseren Verständnis der Idee und der Implementierung in den nachfolgenden Kapiteln werden die wichtigsten Technologien und Werkzeuge an dieser Stelle vorgestellt.

3.1. Eclipse

Das Eclipse Projekt[1] ist eine Open Source Entwicklungsumgebung und wurde 2001 von IBM[2] als solche freigegeben. Es wird zur Entwicklung verschiedenster Anwendungen in der Programmiersprache Java genutzt. Die Eclipse IDE ist ebenfalls in Java programmiert.

Eclipse bietet eine Rich Client Platform (RCP) an. Basierend auf dem Eclipse-Framework lassen sich so verschiedenste Anwendungen entwickeln. Die Entwicklung und Anpassung an die eigenen Aufgabenanforderung erfolgt dabei immer über Plugins. Von der RCP werden dafür die Grundkomponenten, wie die Verwaltung der Laufzeitumgebung, des Dateisystems und die Aktualisierung von Plugins bereitgestellt. Ein Plugin ist ein Erweiterungsmodul, dass von einem Programm zur Laufzeit geladen werden kann um dessen Funktionalität zu erweitern.

Eclipse bietet verschiedene Komponenten zum Umgang mit Quelltexten und anderen Ressourcen an. Diese Komponenten lassen sich den eigenen Vorstellungen und Bedürfnissen anpassen. Beispiele dafür sind Editoren, Sichten (*views*) und Perspektiven.

Editoren sind Fenster zur Darstellung und Bearbeitung von verschiedenen Ressourcen. Zum Programmieren besipielsweise zeigt ein Editor den Quelltext mit Syntaxhervorhebung an. Es gibt aber auch Editoren zur Erstellung von graphischen Benutzeroberflächen mittels Drag-and-Drop. Ein anderes Beispiel sind Baumeditoren zur Darstellung von XML Dateien oder Klassenstrukturen. Sichten dagegen werden zur aufgabenbezogenen Darstellung von Fenstern benutzt. Abhängig vom aktuellen Verwendungszweck können dies beispielsweise Such-, Navigations- oder Debuggingsichten sein. Eine Perspektive schließlich ist eine

[1] www.eclipse.org

[2] www.ibm.com

bestimmte vollständige Zustammenstellung von verschiedenen Editoren, Sichten und Menüleisten, sowie deren konkrete Positionierung. Perspektiven können weitestgehend ebenfalls den eigenen Vorstellungen angepasst werden. Sie können gespeichert und geladen werden. Damit kann sich der Entwickler beispielsweise für jedes Plugin eine individuelle Perspektive erstellen.

3.1.1. Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) ist ein Open Source Framework zur Erstellung, Bearbeitung und Visualisierung von Modellen. Desweiteren stellt das EMF die automatische Quelltextgenerierung aus den Modellen bereit. Ein mit EMF entwickeltes Modell wird *Ecore-Modell* genannt. Die Erstellung eines Ecore-Modells kann mittels eines graphischen oder eines Baumeditors erfolgen. Für ein erstelltes Ecore-Modell wird ein neuer Editor erzeugt, mit dem sich Instanzen des Modells generieren lassen. Diese Instanzen können dann unter anderem weiter manipuliert, abgefragt und im XML Metadata Interchange (XMI) Format serialisiert werden.

Metamodelle Um die abstrakte Syntax eines Modells zu beschreiben, werden in EMF ebenfalls Modelle verwendet, die Metamodelle. Diese Modelle werden in der Sprache Ecore beschrieben.

Die mittels des EMF entwickelten Metamodelle werden durch Klassendiagramme beschrieben. Jede Klasse (*EClass*) wird über einen Namen identifiziert und kann Attribute (*EAttribute*) und Referenzen (*EReference*) enthalten. Ein Attribut hat einen Namen und einen Datentyp (*EDataType*). Eine Referenz verweist auf eine andere Klasse und beschreibt die Assoziation zwischen diesen beiden Klassen. In Abbildung 3.1 ist der Aufbau eines Ecore-Metamodells, sowie die Abhängigkeiten der einzelnen Komponenten schematisch dargestellt.

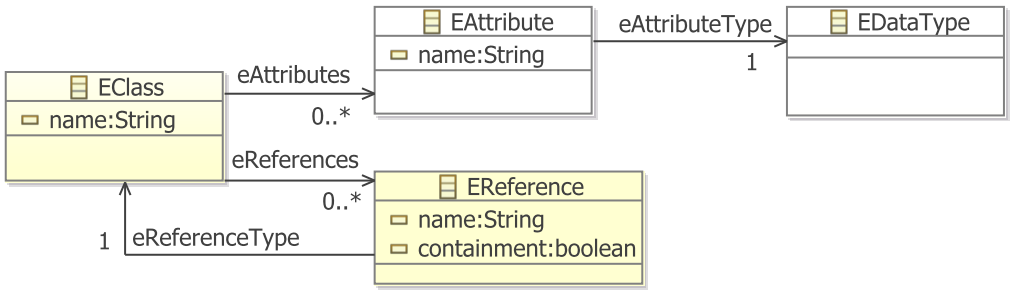


Abbildung 3.1. Schema eines Ecore-Metamodells [Mot09]

3.1.2. Xtext

Xtext[3] basiert auf dem EMF und ist ein Werkzeug zur Entwicklung von Programmiersprachen und Domain Specific Languages (DSLs).

Mit der textuellen Beschreibungssprache von Xtext lässt sich eine Grammatik programmieren. Aus dieser Grammatik erstellt Xtext anschließend automatisch die komplette Infrastruktur für die entwickelte Sprache. Dies beinhaltet neben einem entsprechenden Metamodell auch einen Parser, einen Serialisierer und einen Texteditor. Dabei kann jeder dieser Punkte bei Bedarf an die eigenen Vorstellungen des Entwicklers angepasst werden.

Eine Grammatik kann von bereits existierenden Grammatiken abgeleitet werden. Im Allgemeinen besteht eine Xtext Grammatik aus Regeln (*rules*). Die verwendeten Zeichen am Ende eines Regelaufrufs bestimmen die Kardinalität dieses Aufrufs.

kein Zeichen: exakt einmalige Verwendung

Fragezeichen (?): einmal oder keine

Pluszeichen (+): mindestens einmal

Stern (*): beleibig oft

Desweiteren bestimmt die Art des Zuweisungsoperators die Art des Datums.

Gleichheitszeichen (=): Das Datenfeld ist ein einfacher Wert.

Additionsoperator (+=): Das Datenfeld ist eine Liste.

Boolesche Zuweisung (?=): Das Datenfeld ist ein boolescher Wert.

Listing 3.1 zeigt das Beispiel einer einfachen Xtext-Grammatik. Sie definiert eine neue Sprache, die als einzige Eingabe eine Liste von Begrüßungen akzeptiert. Diese Begrüßungen müssen mit dem Wort Hello beginnen, auf die dann noch ein Name als Identifikator (Identifier (ID)) folgt.

```
1 grammar org.xtext.example.mydsl.MyDsl with
2   org.eclipse.xtext.common.Terminals
3 generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"
4
5 Model:
6   greetings+=Greeting*;
7
8 Greeting:
9   'Hello' name=ID '!';
```

Listing 3.1. Beispiel einer Xtext Grammatik (Xtext Dokumentation)

[3] www.eclipse.org/Xtext

3.1.3. Xtend

Die Programmiersprache Xtend[4] ist ein Dialekt der Programmiersprache Java. Ein Xtend-Programm wird übersetzt in verständlichen, menschenlesbaren Java-Quellcode. Die Vorteile der Sprache Xtend liegen in der kompakteren Syntax und zusätzlichen Funktionalitäten wie Lambdaausdrücken und Operatorüberladung. Listing 3.2 zeigt das „Hello World“ Beispielprogramm in Xtend und kompiliert zu dem Java-Programm, welches in Listing 3.3 dargestellt ist. Ein weiteres besonderes

```
1 class HelloWorld {
2   def static void main(String[] args) {
3     println("Hello World")
4   }
5 }
```

Listing 3.2. Beispiel eines Xtend-Programms (Xtend Dokumentation)

```
1 // Generated Java Source Code
2 import org.eclipse.xtext.xbase.lib.InputOutput;
3
4 public class HelloWorld {
5   public static void main(final String[] args) {
6     InputOutput.<String>println("Hello World");
7   }
8 }
```

Listing 3.3. Generierter Java-Quellcode des Xtend-Beispiels (Xtend Dokumentation)

Merkmal sind die *extension* Methoden. Dies ermöglicht die Erweiterung von bestehenden Typen um weitere Methoden ohne den Typ zu verändern. Das erste Argument einer Extension-Methode muss dabei nicht beim Aufruf mit übergeben werden. Stattdessen kann das Argument die Methode selbst aufrufen. Als ein kleines Beispiel sei hier die Methode `toFirstUpper` der `StringExtensions` genannt. Die herkömmliche Programmierung wäre dafür:

```
StringExtensions.toFirstUpper("hello")
```

Durch Anwendung der Extension-Methode lässt sich diese Programmzeile verkürzen auf:

```
"hello".toFirstUpper()
```

Xtend ist voll kompatibel zu Java und es können alle existierenden Java-Bibliotheken benutzt werden. Desweiteren kann Java- und Xtend-Quellcode beliebig nebeneinander verwendet werden.

[4] www.eclipse.org/Xtend

3.2. Kiel Integrated Environment for Layout Eclipse RichClient

Das Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) Projekt ist ein akademisches Forschungsprojekt, entwickelt am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme des Instituts für Informatik der Christian-Albrechts-Universität zu Kiel. Ziel dieses Projekts ist es, den modellbasierten Entwurf komplexer Systeme stetig zu verbessern. KIELER ist eine Java-basierte Open Source Anwendung unter der Eclipse Public License (EPL)[5]. In KIELER werden viele Modellierungsprojekte aus Eclipse, wie beispielsweise EMF und Xtext, verwendet.

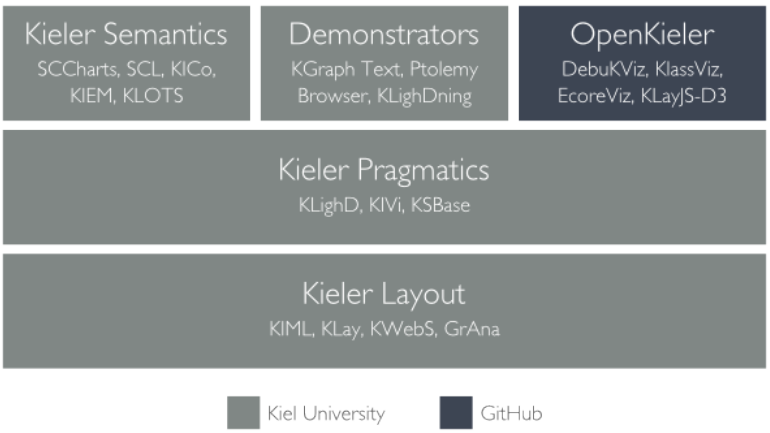


Abbildung 3.2. Das KIELER Projekt (KIELER Dokumentation)

An der Weiterentwicklung des KIELER Projekts sind viele Menschen beteiligt, welche sich oft mit einem bestimmten Anwendungsfall aus einem Teilbereich des gesamten Projektes beschäftigen. Die Struktur des KIELER Projekts ist in Abbildung 3.2 schematisch dargestellt. Die drei Hauptbereiche von KIELER sind die semantische Komponente (*Semantics*), die pragmatische Komponente (*Pragmatics*) und die Layoutkomponente.

Layout Dieser Bereich spiegelt eine der Grundideen des KIELER Projekts wieder, nämlich dem Entwickler zeitintensive Arbeitsschritte zu erleichtern oder ganz abzunehmen. Dies betrifft insbesondere die automatische Anordnung von Elementen in einer graphischen Ansicht eines Modells. Die Hauptkomponente im Layoutbereich stellt die KIELER Infrastructure for Meta Layout (KIML) dar. Diese Infrastruktur stellt die Verbindung zwischen graphischen Editoren und automatischen Layoutalgorithmen dar. Diese Algorithmen werden in KIELER im Paket KIELER Layout Algorithms (KLay) bereitgestellt.

[5] http://rtsys.informatik.uni-kiel.de/~kieler/epl-v10.html

Pragmatik Die pragmatische Komponente beschäftigt sich mit den Aspekten, die die Produktivität des modellbasierten Entwurfs erhöhen. Dies erfolgt durch die an die jeweilige Aufgabenstellung, sowie die Wahrnehmung und Arbeitsweise des Menschen, angepassten Visualisierungen und zur Verfügung gestellten Werkzeuge.

Semantik Die semantische Komponente beschäftigt sich mit der Semantik und Übersetzung von Programmiersprachen, sowie mit Werkzeugen zur Demonstration dieser Arbeitsschritte. So wird beispielsweise die Infrastruktur zur Übersetzung und Ausführung eines Metamodells zur Verfügung gestellt. Die SCCharts sind diesem Bereich zugeordnet und werden im Abschnitt 3.2.1 näher vorgestellt. Desweiteren bietet dieser Bereich verschiedene Ansätze und Werkzeuge zur Simulation von Modellen an.

3.2.1. Sequentially Constructive Charts

Sequentially Contructive Charts (SCCharts) [vHDM+14] ist eine graphische Modellierungssprache zur Entwicklung sicherheitskritischer Echtzeitsysteme. Diese Sprache verwendet eine Statechart-Notation, welche denen der Statecharts von Harel [Har87] ähnelt. Weiterhin bieten SCCharts deterministische Nebenläufigkeit

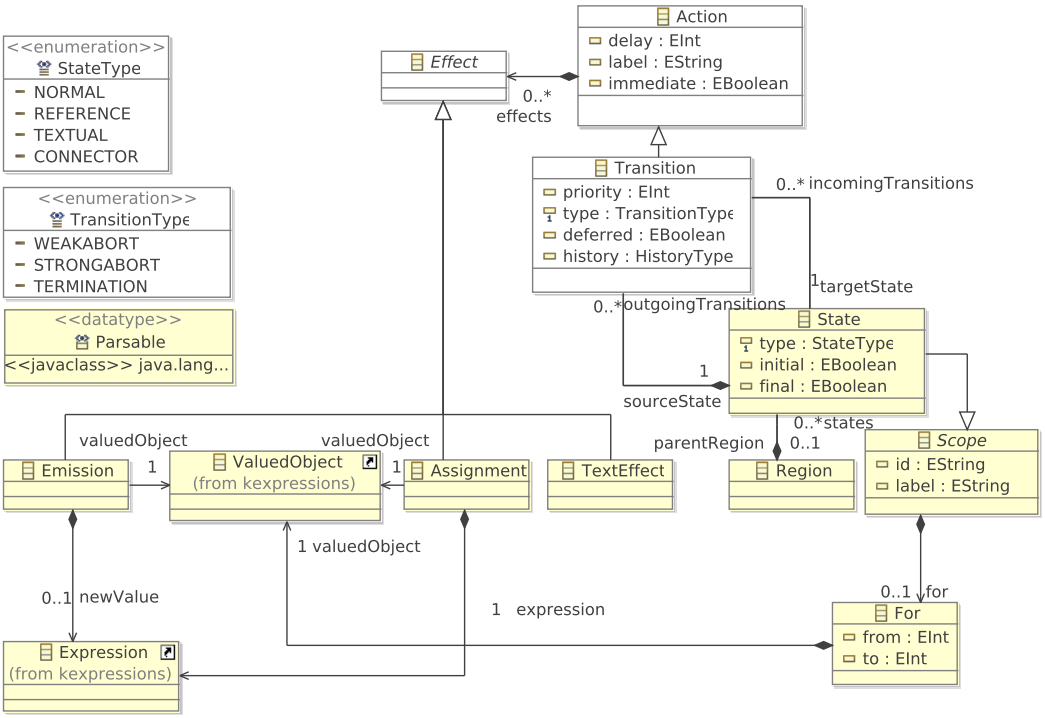
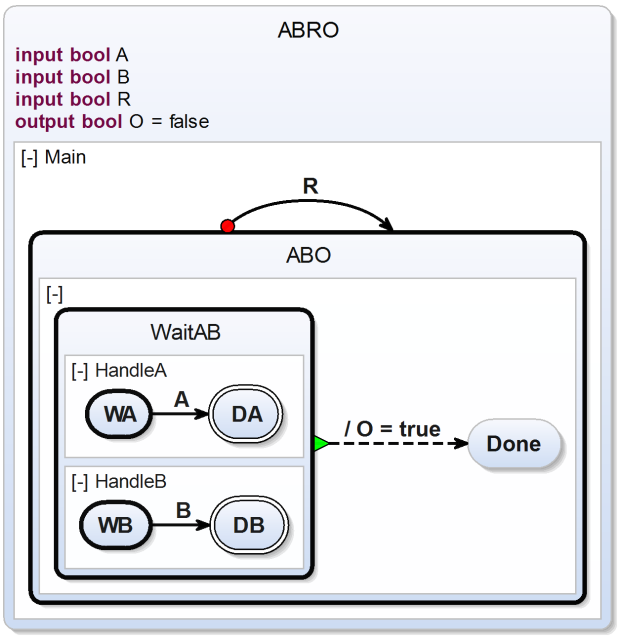


Abbildung 3.3. Das SCCharts Metamodell [MSvH14]

und basieren auf dem in Kapitel 2 vorgestellten SCMoC. Die zugrundeliegende abstrakte Syntax eines SCChart ist in KIELER als EMF Metamodell definiert. Dieses Metamodell ist in Abbildung 3.3 dargestellt.

SCChart Text Die Modellierung eines SCChart erfolgt in der KIELER Implementierung mit der textuellen SCChart Sprache SCT. Diese textuelle Modellierungssprache ist in Form einer Xtext-Grammatik in KIELER definiert. Es folgt eine kurze Einführung in die Sprache SCT. Anhand des Beispielprogramms „ABRO“, welches in Abbildung 3.4 dargestellt ist, werden einige der wichtigsten Elemente vorgestellt. ABRO ist das „Hello World!“ Programm der synchronen Sprachen.

```
1  scchart ABRO {
2    input bool A;
3    input bool B;
4    input bool R;
5    output bool O = false;
6    region Main:
7      initial state ABO {
8        initial state WaitAB {
9          region HandleA:
10           initial state WA
11           ---> DA with A;
12           final state DA;
13         region HandleB:
14           initial state WB
15           ---> DB with B;
16           final state DB;
17       }
18       >--> Done with / O=true;
19       state Done;
20     }
21     o--> ABO with R;
22 }
```



(a) Textuell modelliert in SCT **(b)** ABRO visualisiert als SCChart

Abbildung 3.4. Das ABRO Beispiel (KIELER Dokumentation)

In der ersten Zeile des Programms wird ein SCChart mit der ID ABRO definiert. Die nächsten drei Zeilen enthalten jeweils die Deklaration einer Eingangsvariablen vom Typ bool. Zeile vier beinhaltet die Deklaration einer Ausgangsvariablen O, welche mit dem Wert false initialisiert wird. Nebenläufige Regionen werden in SCCharts mit dem Schlüsselwort region definiert, wie es in Zeile sechs, neun und 13 zu sehen ist. Jede Region muss mindestens einen Zustand (state) beinhalten, davon muss exakt einer der Startzustand sein. Der Startzustand wird mit dem Schlüsselwort initial gekennzeichnet. Das Ende einer Zustandsdefinition wird

mit einem Semikolon gekennzeichnet. Das interne Verhalten eines Zustandes kann wiederum durch nebenläufige Regionen beschrieben werden, wie zum Beispiel innerhalb des Zustandes `WaitAB`. Transitionen müssen vor dem Ende eines Zustandes definiert werden, wie zum Beispiel in Zeile elf zu sehen ist. Sie werden immer von dem ausgehenden Zustand aus gesehen beschrieben. Transitionen können einen *Trigger* und *Effekte* haben, welche durch einen Schrägstrich (/) getrennt werden. Ein Trigger bestimmt, wann die Transition ausgeführt wird. Die Effekte geben an, welche Befehle beim Zustandsübergang ausgeführt werden sollen.

Eine ausführliche Beschreibung der weiteren Funktionen liefert der Technische Bericht zu SCCharts [vHDM+13] und die Online-Dokumentation von KIELER[6].

Core SCCharts und Extended SCCharts Die Schlüsselfunktionen von SCCharts werden über eine kleine Menge von Elementen definiert, den *Core SCCharts*. Diese beinhalten die Beschreibung von hierarchischen Zustandsautomaten und Nebenläufigkeit. Extended-SCCharts hingegen beinhalten eine Fülle von weiteren Elementen, wie zum Beispiel verschiedene *abort* Varianten und Signale. In Abbildung 3.5 ist eine Übersicht der Elemente der Core-SCCharts (oberer Bereich) und der Extended-SCCharts (unterer Bereich) dargestellt. Alle Elemente der Extended-SCCharts können in eine Kombination semantisch äquivalenter Elemente der Core-SCCharts übersetzt werden. Der große Vorteil der Extended-SCCharts liegt darin, dass der Entwickler die Möglichkeit hat, komplexe Zusammenhänge in einem kompakten und einfach lesbaren Modell auszudrücken.

Modelltransformationen Alle Elemente der Extended-SCCharts können über eine Reihe von Model-to-Model (M2M) Transformationen in Core-SCCharts übersetzt werden. Auch die weiteren Übersetzungen bis hin zu ausführbarem sequentiellen C-Code erfolgen über Modelltransformationen [MSvH14]. Eine Übersicht dieser Transformationskette ist in Abbildung 3.6 zu sehen.

Die Transformationskette gliedert sich in zwei Teile, den high-level und den low-level Teil. Die high-level Transformationen beinhalten zunächst die Expandierung der Extended-SCCharts in äquivalente Core-SCCharts. Danach erfolgt die Normalisierung dieser Core-SCCharts. Diese Normalisierung erlaubt nur bestimmte Muster, wie Zustände miteinander verbunden sind, verändert aber keineswegs die Semantik der Core-SCCharts. Diese normalisierten Core-SCCharts können dann anschließend dierekt auf den SC Graph (SCG) abgebildet werden. Der low-level Teil beinhaltet die Transformation von einem SCG in VHDL oder C-Code. Für die low-level Transformation gibt es zwei Ansätze. Die erste Variante ist der Datenfluss-Ansatz, der eine Alternative zu den Esterel *circuit semantics* [PBEB07] darstellt. Die zweite Variante ist der prioritätenbasierte Ansatz,

[6] `http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Textual+SCCharts+Language+SCT`

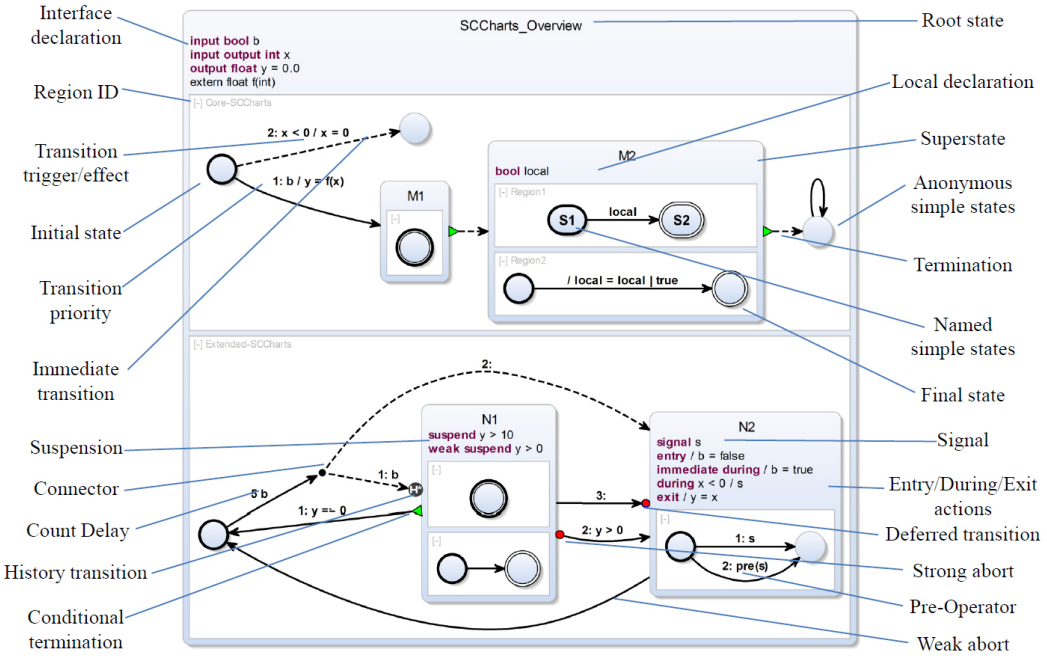


Abbildung 3.5. Übersicht der SCCharts Komponenten. Der obere Bereich enthält nur Elemente der Core-SCCharts, der untere Bereich enthält zusätzliche Elemente der Extended-SCCharts [vHDM+14].

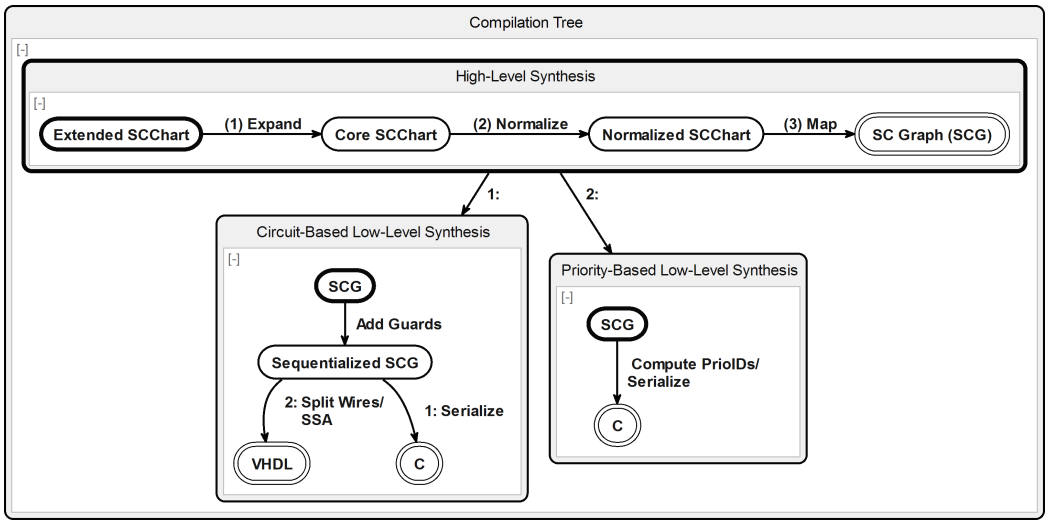


Abbildung 3.6. Vollständige Transformationskette von Extended-SCCharts zu VHDL oder C-Code [MSvH14].

welcher zur besseren Softwaresynthese (Generierung von C-Code) entwickelt wurde [vHDM+14].

Die hier beschriebene Transformationskette ist implementiert im KIELER Compiler (KiCo) Projekt. Der KIELER Compiler ermöglicht die Einbindung von M2M Transformationen von *EObjects* des EMF. Diese können dann schrittweise ausgeführt werden und dienen zur Validierung und Evaluierung der Modelltransformationen.

3.2.2. KIELER Expressions

Ein mächtiges Instrument zur Beschreibung von Variablenzuweisungen und Operationen sind die KIELER Expressions (KExpressions). Neben einer simplen Variablenzuweisung mittels eines *valuedObjects*, können auch komplexe Operationen damit ausgedrückt werden. Diese Operationen umfassen logische Operatoren (*and*, *or*, etc.), arithmeitsche Operatoren (*add*, *sub*, etc.) und relative Operatoren (>, <, etc.). Desweiteren ist eine beliebige Kombination dieser Operatoren möglich. Das KExpression Metamodell ist in Abbildung 3.7 dargestellt.

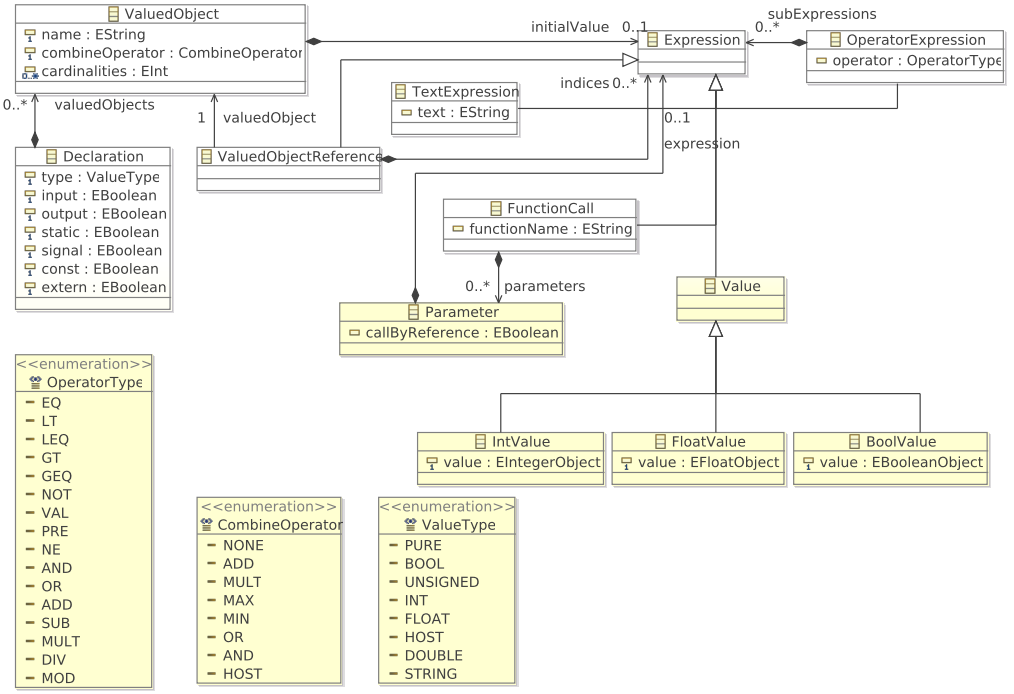


Abbildung 3.7. Das KExpressions Metamodell (KIELER Repository)

SCCharts verwenden die KExpressions zur Beschreibung von Zuweisungen (*Assignment*) der Effekte einer Transition, wie in Abbildung 3.3 ersichtlich. Des Weiteren werden auch die Deklarationen (*Declaration*) aus dem KExpression Metamodell bei der Modellierung von SCCharts wiederverwendet um Ein- und Ausgangs- und lokale Variablen zu definieren.

3.2.3. KIELER Lightweight Diagrams

Die Visualisierung von SCCharts in KIELER erfolgt mittels der KIELER Lightweight Diagrams (KLighD). Dieses Framework verfolgt den Ansatz der *transient views* nach Schneider et. al [SSvH12] und ermöglicht die Darstellung von graphenbasierten Modellen [SSvH13]. Mit diesem Ansatz wird die graphische Repräsentation eines Modells interaktiv erstellt, während der Entwickler beispielsweise an der textuellen Beschreibung arbeitet. In KLighD kann der automatische Layoutalgorithmus KLay Layered verwendet werden. Dieser wurde von Spönemann vorgestellt [Spö09] und von Schulze weiter optimiert [Sch11]. Der Layoutalgorithmus unterstützt hierarchische Knotenplazierung, Ports und stellt eine Fülle, vom Benutzer anpassbaren, Optionen bereit.

Kapitel 4

Datenfluss in SCCharts

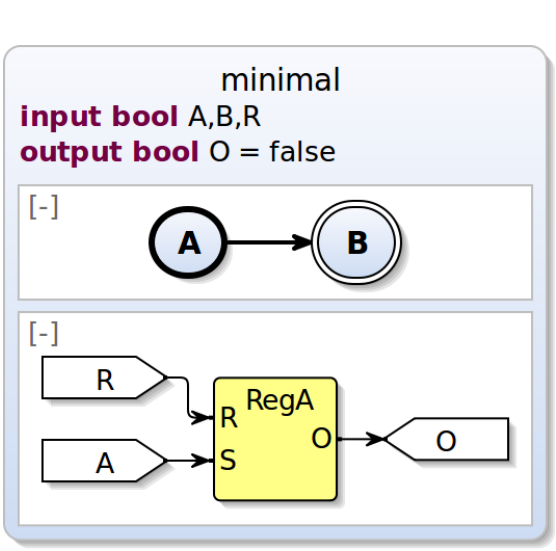
Dieses Kapitel beschreibt das Konzept zur Erweiterung von SCCharts um Datenfluss. Zunächst erfolgt kurz die Vorstellung einer ersten Idee, die referenzierte SCCharts verwendet und in abgewandelter Form in meinem Konzept enthalten bleibt. Danach erfolgt in Unterkapitel 4.1 die Beschreibung der gewünschten Notation von Datenfluss in SCCharts. In Unterkapitel 4.2 werden danach die weiteren Elemente des Modellierungskonzeptes erläutert. Das Unterkapitel 4.3 beschreibt das Konzept zur Visualisierung der Datenflussstrukturen mittels KLighD. Da alle Erweiterungen für den Datenfluss als neue Elemente der Extended SCCharts eingeführt werden, ist eine Anpassung der Modelltransformation nach Core SCCharts notwendig. Unterkapitel 4.4 beschreibt das Konzept dieser Erweiterung der Transformation.

Ein erster Ansatz zur Erweiterung von SCCharts um Datenfluss ist die Verwendung von referenzierten SCCharts. Die Referenzen erlauben den Zugriff auf ein separates SCChart als Einbindung in ein anderes SCChart. Dieser erste Ansatz erlaubt die Referenzierung in einer neuen nebenläufigen Umgebung, in der referenzierte SCCharts als Aktor eines Datenflussblockdiagramms dargestellt werden. Diese neue Klasse einer Umgebung heißt `Dataflow` und enthält neben diesen Aktoren (im Folgenden auch *Referenzknoten* genannt, da sie ein separates SCChart referenzieren) auch Knoten für Eingangs- und Ausgangswerte. Mit diesen drei einfachen Knotentypen lässt sich ein klassisches Datenflussdiagramm erstellen, das Eingangswerte, einen Aktor zur Verarbeitung eben dieser, und Ausgangswerte enthält. In Abbildung 4.1a[1] ist ein kleines Beispiel von einem SCChart mit zwei nebenläufigen Regionen zu sehen. Die obere Region enthält ein kleines Kontrollflussdiagramm und die untere Region enthält ein Datenflussblockdiagramm. Die Datenflussregion enthält ein separat als SCChart modelliertes Flipflop (siehe Abbildung 4.2[1]), welches hier referenziert wird.

Dagegen ist es mit meinem Lösungsansatz möglich, dass Datenfluss nicht nur mittels referenzierter SCCharts modelliert werden kann. Zusätzlich ist eine direkte Modellierung von Datenfluss in einer entsprechenden Umgebung möglich. Desweiteren können auch eigens definierte Knoten mit Daten- oder Kontrollfluss beschrieben und aufgerufen werden. Die Modellierung von Datenfluss erfolgt textuell mit der Sprache SCT, welche als Xtext Grammatik im KIELER Projekt

[1]Quelle: KIELER Projekt Wiki: `http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Home`

definiert ist. Auf die einzelnen Aspekte des gesamten Konzeptes gehe ich in den folgenden Unterkapiteln, beginnend bei der Notation, ein.



(a) Graphische Darstellung

```
1  scchart minimal {
2    input bool A, B, R;
3    output bool O = false;
4    region:
5      initial state A
6      ——> B;
7      final state B;
8    dataflow:
9      node A "A" input: A
10     ——> RegA.S;
11     node R "R" input: R
12     ——> RegA.R;
13     node RegA "RegA": flipflop
14     O ——> O;
15     node O "O" output: O;
16 }
```

(b) Textuelle Modellierung in SCT

Abbildung 4.1. SCChart mit Daten- und Kontrollfluss (KIELER Wiki)

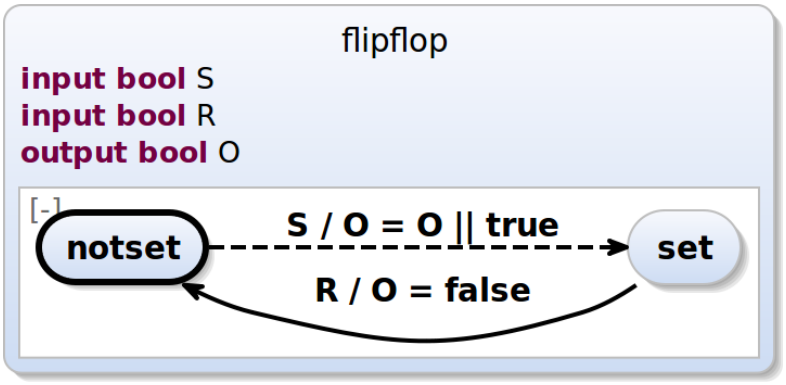


Abbildung 4.2. RS Flipflop als SCChart modelliert (KIELER Wiki)

4.1. Notation von Datenfluss in SCCharts

Für das eben vorgestellte Beispiel wurde noch die für Datenfluss eher umständliche Notation im Stil der SCCharts, beziehungsweise generell von Kontrollflussdigrammen verwendet. Jede Transition wird textuell vom Ausgangspunkt zum Zielpunkt beschrieben, wie in der Abbildung 4.1b zu sehen ist. Diese Notation ist

ungewöhnlich für Datenfluss und kann vom Modellierungsumfang sehr schnell sehr mächtig und unübersichtlich werden. Als Beispiel zur Verdeutlichung kann man eine sehr einfache Zuweisung $x = a + b$ betrachten, siehe Abbildung 4.3b. Verwendet man hierfür die klassische SCChart-Notation (siehe Abbildung 4.3a) so würde man ohne die Definierung der Knoten für die beiden Eingangswerte, den Ausgangswert und den Aktor (die Addition) bereits drei Befehle zur Modellierung benötigen. Das sind genau betrachtet zwei Befehle für den Datenfluss von den beiden Eingangswerten zum Aktor und eine dritte Befehlszeile vom Ausgang des Aktors zum Ausgangsknoten.

Für mein Konzept wird zur Vereinfachung der Modellierung von Datenfluss die Notation angepasst. Dabei gilt, dass die Beschreibung nicht mehr von den Eingangswerten aus betrachtet wird, sondern von den Ausgangswerten. SCCharts beinhalten die in Abschnitt 3.2.2 vorgestellten KExpressions. Damit ist es bereits möglich komplexe Operationen und Variablenzuweisungen zu beschreiben und die Notation erfolgt ebenfalls von den Ausgangswerten aus betrachtet. Dies wird hier ausgenutzt. In Abbildung 4.3 ist auf der linken Seite das eben beschriebene Beispiel in der herkömmlichen Notation zu sehen, und auf der rechten Seite in der Notation, wie sie in meinem Konzept verwendet wird.

```
a --> add.in1;
b --> add.in2;
add.out --> x;
```

(a) Ursprüngliche Notation

```
x = a + b;
```

(b) Neue Notation

Abbildung 4.3. Beispiel zur Notation von Datenfluss in SCCharts

4.2. Modellierungskonzept von Datenfluss

Der nächste Schritt zur Erweiterung von SCCharts um Datenfluss ist die Anpassung des zugrundeliegenden Metamodells (siehe Abbildung 3.3) und der dazugehörigen Grammatik der textuellen Progammiersprache SCT. Dieses Unterkapitel erläutert die Ideen und das weitere Modellierungskonzept.

4.2.1. Die Datenflussumgebung

Die Modellierung von Datenfluss erfolgt wie schon beim Kontrollfluss in SCCharts innerhalb nebenläufiger Regionen. Zunächst stellt sich dabei die Frage, ob für beide Modellierungskonzepte ein separater Typ einer Region verwendet werden soll, oder ob eine Region für beide Konzepte gültig ist. Für mein Konzept habe ich mich zur Verwendung von zwei unterschiedlichen Typen entschieden. Somit gibt es einmal die klassische Region für die Modellierung von Kontrollfluss und einmal die Datenflussregion für die Datenflussmodellierung. Beide Typen können

beliebig parallel verwendet werden. Der Vorteil dieser Trennung ist zunächst einmal, dass die bisherige Modellierung von SCCharts unagetastet und weiterhin vollständig kompatibel zum KIELER Compiler bleibt. Für die Datenflussmodellierung bietet diese Trennung weiterhin den Vorteil, dass mögliche weiterführende und ergänzende Arbeiten einfacher hinzugefügt werden können.

Die neue Umgebung zur Modellierung von Datenfluss heißt Dataflow. Damit die bereits existierende Klasse Region weiterhin bestehen bleibt und zusammen mit der neuen Klasse Dataflow beliebig nebenläufig verwendet werden kann, ist eine neue Oberklasse im Metamodell notwendig. Das ist die Klasse Concurrency. In Abbildung 4.4 ist die Einbettung der Datenflussumgebung im Metamodell der SCCharts dargestellt.

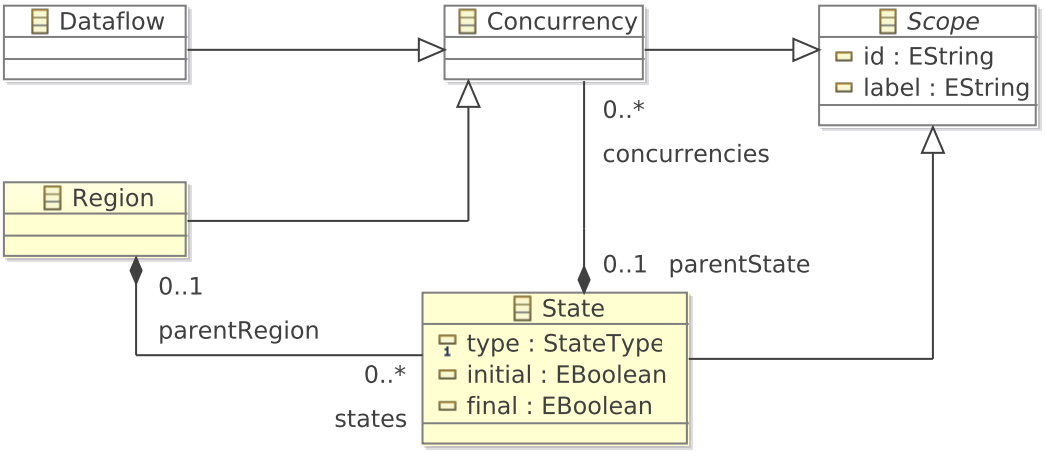


Abbildung 4.4. Einbettung der Datenflussumgebung im SCChart Metamodell (Ausschnitt)

Die Regel der Xtext-Grammatik für die Verwendung dieser Klasse ruft die entsprechende Regel für eine Region oder eine Datenflussumgebung auf. Dies ist in Listing 4.1 dargestellt. Ein Zustand (State) kann beliebig viele Instanzen der Klasse Concurrency enthalten.

```
1 Concurrency returns scccharts::Concurrency:
2   Region | Dataflow
3 ;
```

Listing 4.1. Grammatikregel der Klasse Concurrency

In einer Datenflussumgebung können beliebig viele Gleichungen (Equations) zur direkten Datenflussmodellierung und Knoten (Nodes) enthalten sein. Die direkte Modellierung von Datenfluss, ohne Einbettung in einen Knoten, entspricht einer Gleichung der Form $x = e$, wobei x eine Variable und e ein Ausdruck zur Berechnung dieser Variablen ist. Im nachfolgenden Abschnitt 4.2.2 wird das Konzept

und die Verwendung der Datenflussgleichungen näher betrachtet. Die Knoten sind unterteilt in drei Typen, die definierenden Knoten (DefineNode), die aufrufenden Knoten (CallNode) und die referenzierenden Knoten (ReferenceNode), welche alle von der Oberklasse Node abgeleitet sind. Die Modellierung und Verwendung dieser Knoten wird in Abschnitt 4.2.3 genauer erläutert. In Abbildung 4.5 ist der Ausschnitt zur Einbindung dieser Komponenten in einer Datenflussumgebung im Metamodell der SCCharts zu sehen.

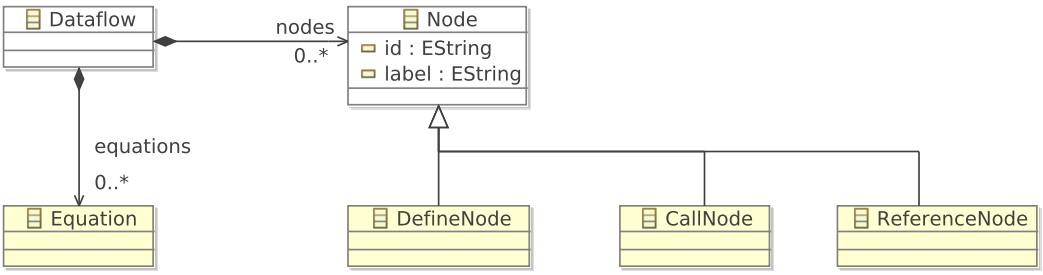


Abbildung 4.5. Komponenten der Datenflussmodellierung im SCChart Metamodell (Ausschnitt)

Entsprechend zu dem Metamodell ist in Listing 4.2 der dazugehörige Abschnitt aus der Xtext-Grammatik der SCCharts dargestellt. Die textuelle Modellierung einer Datenflussumgebung beginnt mit dem Schlüsselwort dataflow. Gefolgt von einem optionalen Bezeichner (Identifier (ID)) und einem optionalen Label. Der Abschnitt ('['for=For']')? aus Zeile vier des Listings dient zur Deklaration von Arrays in SCCharts und ist ebenfalls optional. Mittels der danach folgenden Deklarationen lassen sich lokale Variablen definieren. Anschließend erfolgt die Beschreibung der Knoten und Datenflussgleichungen in beliebiger Anzahl und Reihenfolge.

```
1  Dataflow returns scccharts::Dataflow:
2    {scccharts::Dataflow}
3    (annotations += Annotation)*
4    'dataflow' (id=ID)? (label=STRING)? ('['for=For']')? ':'
5    (declarations+=Declaration)*
6    (
7      (equations += Equation)
8      |
9      (nodes += Node)
10   )*
11 ;
```

Listing 4.2. Grammatikregel der Datenflussumgebung

4.2.2. Direkte Datenflussmodellierung

Innerhalb einer Datenflussregion ist es möglich einer globalen oder lokalen Variable einen Ausdruck zuzuweisen. Diese Zuweisung kann entweder ein anderer Variablenwert, eine Konstante oder eine komplexe Berechnung sein. Dies erfolgt über die Verwendung der KExpressions, welche bereits viele für Datenfluss gängige Operationen möglich machen. So beinhalten sie beispielsweise arithmetische, relative und logische Operatoren. Durch diese einfache Beschreibung komplexer Operationen lassen sich berechnungsintensive Abläufe in einer Datenflussumgebung einfach modellieren.

Eine Gleichung der direkten Datenflussmodellierung besitzt die folgenden drei Referenzen. Zunächst einmal ein `ValuedObject`, das ist die Variable, der ein Wert zugewiesen wird. Desweiteren einen Ausdruck (`Expression`), welcher diesen Wert berechnet. Und eine Referenz `node`, welche auf einen Knoten verweisen kann. Diese dritte Referenz ist dabei optional. In Abbildung 4.6 ist der Ausschnitt des Metamodells zu den Datenflussgleichungen dargestellt. Die Klassen

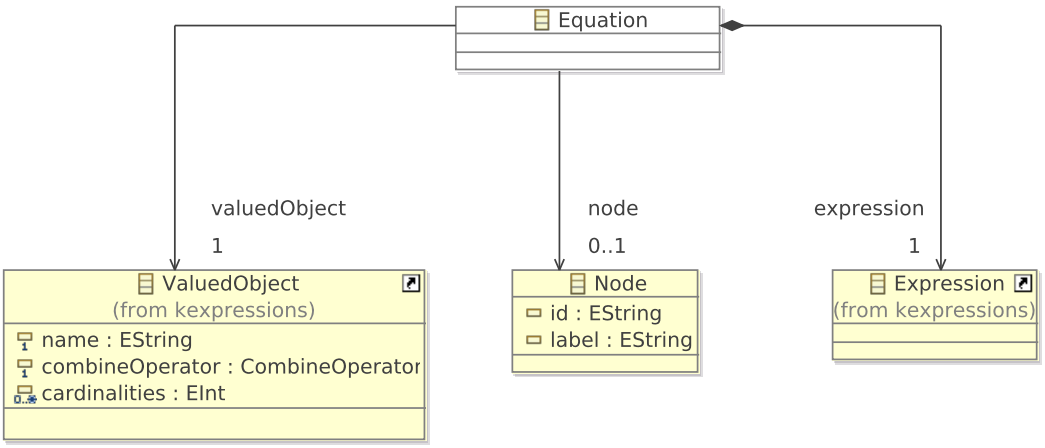


Abbildung 4.6. Datenflussgleichung im SCChart Metamodell (Ausschnitt)

`ValuedObject` und `Expression` entsprechen dabei denen aus dem KExpressions Metamodell. Listing 4.3 zeigt die Grammatikregel zur Beschreibung einer Datenflussgleichung. Dabei ist Folgendes zu beachten. Verwendet man die optionale Referenz `node`, muss diese auf einen definierenden oder aufrufenden Knoten verweisen. Der Ausdruck in dieser Datenflussgleichung ist dann zwingend ein Ausgangswert dieses Knotens. Der Ausgangswert entspricht dabei einer Variablenreferenz (`ValuedObjectReference`), welche von der Klasse `Expression` abgeleitet ist. Die textuelle Modellierung eines solchen Zugriffs ist dann beispielsweise `x = flipflop.o`, wobei `x` eine Variable, `flipflop` ein Knoten und `o` ein Ausgangswert dieses Knotens ist.

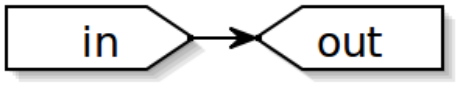
```
Equation returns scccharts::Equation:
  {scccharts::Equation}
  (
    (valuedObject=[kexpressions::ValuedObject])'='(expression=Expression)';'
    |
    (valuedObject=[kexpressions::ValuedObject])'='node=[scccharts::Node|ID]
                                          '.'(expression=ValuedObjectReference)';'
  )
;
```

Listing 4.3. Grammatikregel einer Datenflussgleichung

Die Modellierung von Datenflussgleichungen sei im Folgenden exemplarisch an zwei Beispielen verdeutlicht. In Abbildung 4.7 ist ein einfaches Beispiel einer Datenflussumgebung mit je einer lokalen booleschen Eingangs- und Ausgangsvariable zu sehen. In diesem Beispiel besteht der Ausdruck zur Berechnung von out lediglich aus der Zuweisung des Eingangswertes in. Auf der linken Seite ist der Abschnitt aus der textuellen Modellierung, auf der rechten Seite die graphische Darstellung zu sehen.

```
dataflow:
  input bool in;
  output bool out;

  out = in;
```



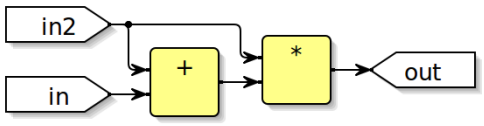
(a) Modellierung in SCT **(b)** Visualisierung mit KLighD

Abbildung 4.7. Beispiel einer Datenflussgleichung (Zuweisung)

Statt einer einfachen Zuweisung kann der Ausdruck auch aus einer Berechnung bestehen. In Abbildung 4.8 ist die textuelle Modellierung der Datenflussgleichung out = (in + in2) * in2, sowie deren graphische Repräsentation zu sehen.

```
dataflow:
  input int in, in2;
  output int out;

  out = (in + in2) * in2;
```



(a) Modellierung in SCT **(b)** Visualisierung mit KlighD

Abbildung 4.8. Beispiel einer Datenflussgleichung (Berechnung)

Auf die Modellierung von Datenflussgleichungen, die auf einen Knoten verweisen wird im folgendem Abschnitt nach der Vorstellung dieser Knoten eingegangen.

4.2.3. Knoten als Datenflussaktoren

Neben der direkten Modellierung von Datenfluss ist auch die Verwendung beliebig vieler Knoten innerhalb einer Datenflussumgebung möglich. Diese Knoten dienen zur Definition und zum Aufruf verschiedener Datenflusskomponenten und -modelle. Es gibt drei Typen von Knoten:

Definierende Knoten: Ein Knoten zur späteren (mehrfachen) Verwendung wird definiert. Dabei werden Anzahl und Typ der Eingänge und Ausgänge festgelegt. Wie bei der direkten Modellierung in einer Datenflussregion kann den Ausgängen ein komplexer Ausdruck vom Typ `Expression` zugewiesen werden. Alternativ zum Datenfluss kann auch eine beliebige Anzahl von Zuständen hinzugefügt werden, welche dann wiederum Kontrollfluss definieren.

Aufrufende Knoten: Dieser Knotentyp verweist auf einen zuvor definierten Knoten. Im Gegensatz zu den definierenden Knoten werden alle Aufrufe von diesen Knoten später visualisiert und gegebenenfalls transformiert werden.

Referenzierende Knoten: Diese Knoten stellen eine Sonderform der aufrufenden Knoten dar, in dem sie ein separates (nicht in der gleichen Datei beschriebenes) SCChart aufrufen.

Mit dieser Unterscheidung bei den Knoten ist es möglich das Verhalten einer Datenflusskomponente einmalig zu definieren, aber beliebig oft im Modell zu verwenden, ohne dass es erforderlich ist, das Verhalten jedesmal neu beschreiben zu müssen.

In Abbildung 4.9 ist ein detaillierter Ausschnitt der Einbindung der Klasse `Node` im Metamodell der SCCharts zu sehen. Dargestellt ist die Oberklasse `Node` und die drei davon abgeleiteten Klassen der verschiedenen Knotentypen, sowie deren Attribute und Referenzen. Entsprechend dazu ist in Listing 4.4 die dazugehörige Grammatikregel der Klasse `Node` zu sehen. Diese besteht aus dem Aufruf der jeweiligen Regel zur Beschreibung einer der drei möglichen Knotentypen.

```
1 Node returns scchart::Node:
2   ReferenceNode | CallNode | DefineNode
3 ;
```

Listing 4.4. Grammatikregel der Klasse Node

In Listing 4.5 ist der konkrete Abschnitt aus der Grammatik zur Beschreibung eines definierenden Knoten dargestellt. Auf das Schlüsselwort `node` folgt ein Bezeichner. Danach folgt eine beliebigen Anzahl von Variablendeklarationen, welche die benötigten Eingangswerte für das im Knoten beschriebene Verhalten repräsentieren. Nach dem nächsten Schlüsselwort `returns` folgen erneut Variablendeklarationen, welche die Ausgänge des Knotens bestimmen. Innerhalb von

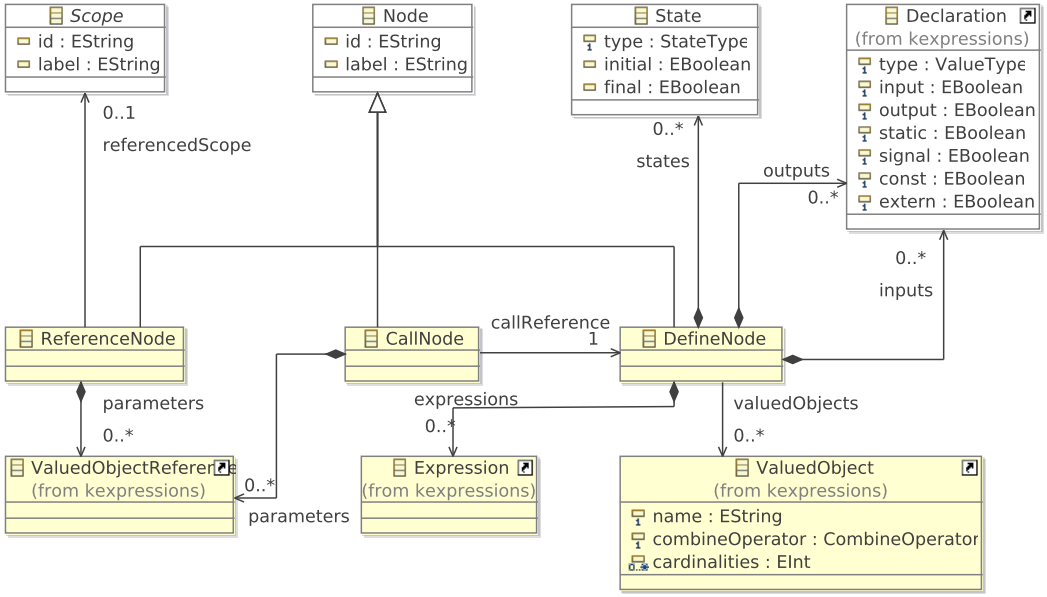


Abbildung 4.9. Datenflussknoten im SCChart Metamodell (Detailausschnitt)

geschweiften Klammern kann nun das Verhalten beschrieben werden. Für die Modellierung von Datenfluss kommt erneut die Syntax der Datenflussgleichungen der Form $x = e$ zum Einsatz. Die Variable x entspricht einem ValuedObject und der Ausdruck e einer Expression aus dem KExpressions Metamodell. Alternativ kann in einem definierenden Knoten auch Kontrollfluss modelliert werden. Dies erfolgt über die Beschreibung beliebig vieler Zustände (States). Um das in einem definierenden Knoten beschriebene Verhalten zu verwenden, werden die aufrufenden Knoten benutzt. Beliebig viele aufrufende Knoten können auf den gleichen definierenden Knoten verweisen. Auf diese Weise muss das

```
1  DefineNode returns scccharts::DefineNode:
2    {scccharts::DefineNode}
3    'node' (id=ID) '(' (inputs+=Declaration)* ')'
4    'returns' '(' (outputs+=Declaration)* ')' '{'
5    (
6      ((valuedObjects += [kexpressions::ValuedObject])
7                 '=' (expressions += Expression) ';')*
8      |
9      (states += State)*
10   )
11   '}'
12 ;
```

Listing 4.5. Grammatikregel eines definierenden Knotens

Verhalten nur einmal beschrieben werden, kann aber an verschiedenen Stellen in der Datenflussumgebung verwendet werden.

Listing 4.6 zeigt die Syntax der aufrufenden Knoten. Diese beginnen ohne ein explizites Schlüsselwort, sondern direkt mit einem Bezeichner. Wie in Zeile drei ersichtlich, muss auf einen definierenden Knoten verwiesen werden. Danach folgen eingeschlossen in runden Klammern die Variablen, die als Parameter mit übergeben werden sollen. Dabei ist zu beachten, dass die übergebenen Parameter in der Reihenfolge übereinstimmen, wie sie beim definierenden Knoten aufgelistet werden. Das heißt, Parameter eins wird auf die erste Variable der Eingangswertdeklarationen des definierenden Knotens gesetzt, Parameter zwei auf die zweite Variable, und so weiter und so fort.

```
1 CallNode returns scharts::CallNode:
2   {scharts::CallNode}
3   (id=ID) '=' callReference = [scharts::DefineNode|ID]
4   '('(parameters += ValuedObjectReference)?
5     (',' parameters += ValuedObjectReference)* ')'
6   ';'
7 ;
```

Listing 4.6. Grammatikregel eines aufrufenden Knotens

Der letzte Knotentyp ist der referenzierende Knoten. Dieser verhält sich ähnlich wie der aufrufende Knoten. Statt eines definierenden Knotens wird aber bei einem referenzierenden Knoten auf ein separates SCChart verwiesen. Im KIELER Projekt ist das Wurzelelement eines SCChart immer eine Instanz der Klasse State, weshalb die Referenz diesen Typ erfordert. Daher unterscheidet sich die Beschreibung der Syntax eines referenzierenden Knotens nur in Zeile drei des Listings 4.7. Hier gibt es zusätzlich das Schlüsselwort ref auf welches dann die ID des Zustandes des zu referenzierenden SCChart folgt. Innerhalb von runden Klammern werden wie bereits erklärt die als Parameter zu übergebenden Variablen festgelegt.

```
1 ReferenceNode returns scharts::ReferenceNode:
2   {scharts::ReferenceNode}
3   (id=ID) (label=STRING)? '=' 'ref' referencedScope = [scharts::State|ID]
4   '('(parameters += ValuedObjectReference)?
5     (',' parameters += ValuedObjectReference)* ')'
6   ';'
7 ;
```

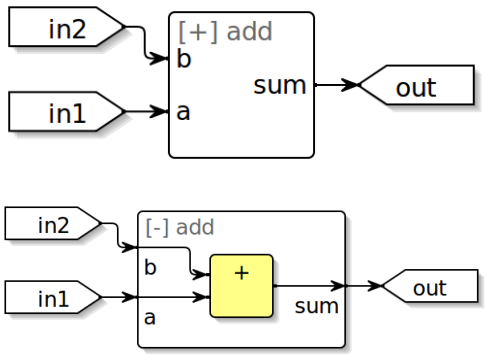
Listing 4.7. Grammatikregel eines referenzierenden Knotens

Es wäre auch denkbar auf eine Umgebung zu verweisen, um das darin beschriebene Verhalten zu referenzieren. Deshalb ist im Metamodell in Abbildung 4.9 eine Referenz auf die Klasse Scope zu sehen. Sowohl die nebenläufigen Umgebungen (Concurrencies) und die Zustände (States) leiten sich von der Klasse Scope ab, was im Ausschnitt des Metamodells in Abbildung 4.4 zu sehen ist. Eine Erweiterung des Konzeptes der Referenzknoten wäre ohne Änderung am Metamodell, sondern ausschließlich über die Grammatik möglich.

Im Folgenden wird die Verwendung der verschiedenen Knotentypen exemplarisch dargestellt. Abbildung 4.10 zeigt auf der linken Seite die Modellierung einer Datenflussumgebung mit den lokalen Variablen in1, in2 und out in SCT. In dieser Umgebung wird weiterhin der Knoten add definiert, in welchem die Summenberechnung von zwei ganzzahligen Werten als Datenfluss beschrieben ist. Die Verwendung dieses definierenden Knoten erfolgt über den aufrufenden Knoten call. Abschließend wird der Ausgangsvariablen out noch der Ausgangswert sum des Knotenaufrufs zugewiesen. Auf der rechten Seite der Abbildung 4.10 ist die Visualisierung dieser Datenflussumgebung dargestellt. Einmal mit kollabiertem aufrufenden Knoten und einmal im expandierten Zustand. Die Idee hinter diesem Prinzip der kollabierten und expandierten Knoten wird im kommenden Unterkapitel 4.3 näher beschrieben.

```
1  dataflow:
2    input int in1, in2;
3    output int out;
4
5    node add(int a, b)
6            returns (int sum) {
7      sum = a + b;
8    }
9
10   call = add(in1, in2);
11   out = call.sum;
```

(a) Modellierung in SCT



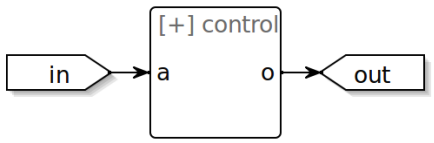
(b) Visualisierung mit KlighD

Abbildung 4.10. Beispiel eines definierenden Knotens, welcher Datenfluss enthält, und dessen Aufruf. Die Visualisierung in (b) zeigt den Knoten einmal kollabiert (oben) und einmal expandiert (unten).

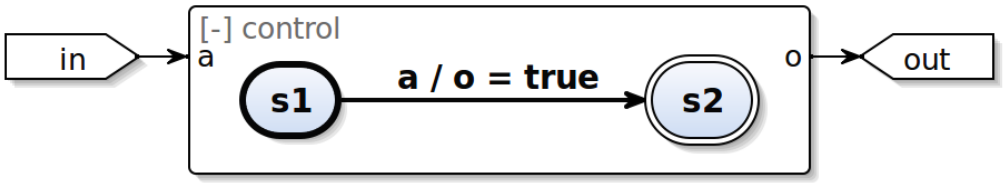
Das nächste Beispiel, was in Abbildung 4.11 dargestellt ist, ist eine Datenflussumgebung, die wieder einen definierenden und einen aufrufenden Knoten enthält. Diesmal enthält der definierende Knoten allerdings Kontrollfluss, wie links oben zu sehen ist. Die Abbildung 4.11 zeigt ebenfalls die Visualisierung dieser Datenflussumgebung sowohl mit kollabiertem, als auch expandiertem aufrufenden Knoten.

```
1  dataflow:
2    input bool in;
3    output bool out;
4
5    node control(bool a) returns (bool o) {
6      initial state s1
7        --> s2 with a / o = true;
8      final state s2;
9    }
10   call = control(in);
11   out = call.o;
```

(a) Modellierung in SCT



(b) Visualisierung mit KlighD (kollabiert)



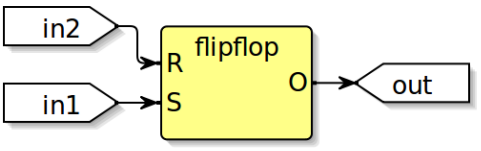
(c) Visualisierung mit KlighD (expandiert)

Abbildung 4.11. Beispiel eines definierenden Knotens, welcher Kontrollfluss enthält, und dessen Aufruf

Die Verwendung der referenzierenden Knoten ist in einem letzten Beispiel dieses Abschnitts dargestellt. In Abbildung 4.12 ist die Modellierung eines referenzierenden Knotens in Zeile fünf des Listings zu sehen. Nach der Deklaration lokaler Variablen, erfolgt der Aufruf eines separaten SCChart, das ist das Flipflop aus Abbildung 4.2. Dieses referenzierte Flipflop ist in einer separaten Datei im gleichen Projektordner gespeichert. Die Variablen `in1` und `in2` werden als Parameter mit übergeben. Abschließend wird der Variablen `out` der Ausgangswert des Flipflops zugewiesen. Auf der rechten Seite ist die dazugehörige graphische Repräsentation dargestellt.

```
1  dataflow:
2    input bool in1, in2;
3    output bool out;
4
5    call = ref flipflop(in1, in2);
6    out = call.O;
```

(a) Modellierung in SCT



(b) Visualisierung mit KlighD

Abbildung 4.12. Beispiel eines referenzierenden Knotens und Aufruf

4.2.4. Verschachtelung von Daten- und Kontrollfluss

SCCharts lassen sich beliebig hierarchisch verschachteln. Ein Zustand eines Kontrollflussmodells kann wiederum beliebig viele parallele Regionen und Zustände enthalten. In meinem Datenflusskonzept ist der hierarchische Aufbau ebenfalls möglich. Die Umgebungen für Daten- und Kontrollfluss können beliebig nebenläufig verwendet werden. Referenzierte SCCharts können weiterhin wie gewohnt eingesetzt werden. Darüber hinaus gibt es mit den Referenzknoten eine weitere Möglichkeit auf ein separates SCChart zu verweisen. Diese referenzierten SCCharts können wiederum beliebig verschachtelten und nebenläufigen Daten- und Kontrollfluss enthalten. Zusätzlich bieten die definierenden Knoten die Möglichkeit entweder Datenfluss oder Kontrollfluss zu modellieren. Durch diese Entscheidungsfreiheit in der Definierung von Knoten mit Kontroll- oder Datenfluss ist es möglich beide Modellierungskonzepte beliebig zu verschachteln. In Abbildung 4.13 ist ein Beispielmodell mit verschachteltem Daten- und Kontrollfluss dargestellt. Das SCChart Hierarchy hat zwei nebenläufige Umgebungen, einmal eine Kontrollflussregion und einmal eine Datenflussregion. Die Kontrollflussregion, welche im unteren Bereiche zu sehen ist, enthält zwei Zustände. Im ersten Zustand sind erneut zwei nebenläufige Regionen enthalten, eine Datenflussumgebung innerDataflow und eine Kontrollflussumgebung innerRegion. Die obere Region, die äußere Datenflussumgebung outerDataflow des SCChart enthält einen definierenden Knoten, welcher einmal aufgerufen wird. Der Knoten ist expandiert dargestellt, so dass die beiden enthaltenen Zustände zu sehen sind. Im zweiten Zustand innerhalb dieses Knotens sind erneut zwei nebenläufige Umgebungen zu sehen. Einmal mit Kontrollfluss in der Umgebung nestedRegion und einmal Datenfluss in der Umgebung nestedDataflow.

4.3. Visualisierung von Datenflussregionen

Da das grundlegende Konzept für die Erweiterung von SCCharts um Datenfluss das vorhandene Metamodell und die Grammatik erweitert, bildet auch die bereits existierende Visualisierung von SCCharts den Ausgangspunkt zur Visualisierung des Datenflusses. Die Diagrammsynthese beschreibt die Visualisierung eines gegebenen SCChart mittels KIELER Lightweight Diagrams. In diesem Abschnitt werden die Ideen zur Darstellung von Datenfluss, sowie einzelner Komponenten erläutert.

Die SCCharts-Sprache folgt dem Konzept des pragmatischen modellbasierten Entwurfs nach Fuhrmann und von Hanxleden [FvH10a]. Das bedeutet, dass besonders bei großen Modellen die praktische Handhabung wie zum Beispiel Wartung und Erweiterung von Modellen im Fokus steht. Eine Schlüsselfunktion stellt dabei das automatische Layout dar, so dass dem Benutzer die zeitaufwändige Arbeit des manuellen Erstellens einer graphischen Repräsentation seines

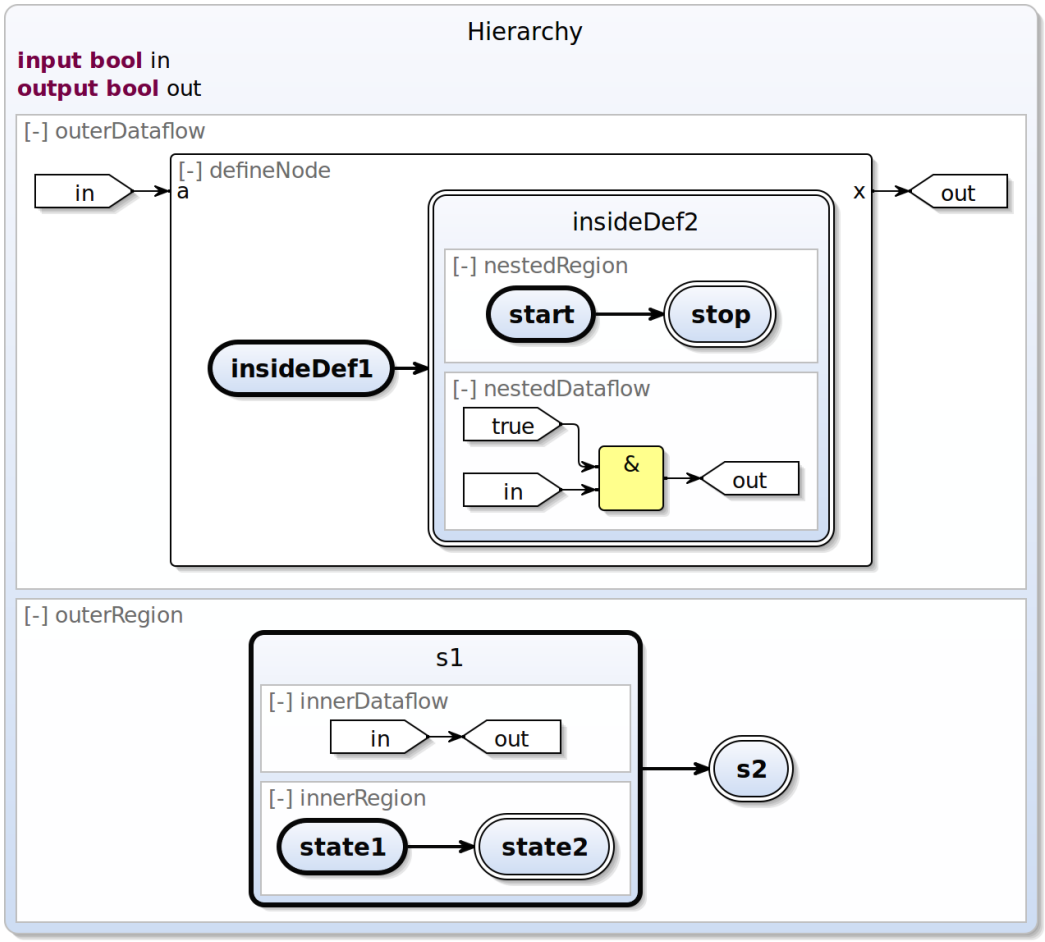
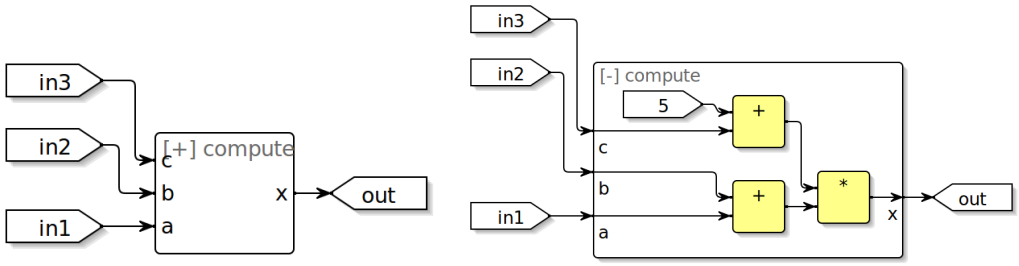


Abbildung 4.13. Beispielmodell zur Verschachtelung von Daten- und Kontrollfluss

Modells abgenommen wird. So wird jede Änderung an einem SCChart nach dem Speichern automatisch im Modell angezeigt. Neben dem automatischen Layout geht es bei der Pragmatik des modellbasierten Entwurfs auch um die Anpassung der Visualisierung. So tragen zum Verständnis und zur Übersichtlichkeit von Modellen auch Benutzeraktionen, wie der Fokus auf eine bestimte Region, einen bestimmten Zustand oder das Hervorheben von ausgewählten Elementen, deutlich bei [FvH10b].

Dieser pragmatische Ansatz wird auch bei den um Datenfluss erweiterten SCCharts angewendet und fortgesetzt. Wie bereits vorgestellt, gibt es verschiedene Knotentypen bei der Modellierung von Datenfluss. Ein definierender Knoten beschreibt das Verhalten einer Datenflusskomponente, wird aber zunächst nicht dargestellt. Die Verwendung kann aber durch die aufrufenden Knoten beliebig oft erfolgen. Bei der Visualisierung wird für jeden Aufruf nur ein Datenflussaktor dargestellt, das konkrete Verhalten wird zunächst ausgeblendet. Dadurch wird

die Übersichtlichkeit des Modells gewahrt. Interessiert man sich für eine größere Detailliertheit, so kann dieses Element expandiert werden um den Inhalt anzeigen zu lassen. Zur Verdeutlichung ist in Abbildung 4.14 ein kleines Datenflussmodell dargestellt mit drei Eingängen, einem Aktor und einem Ausgang. Die definierte Berechnung innerhalb des Aktors compute ist zunächst kollabiert dargestellt, trotzdem ist ersichtlich, dass die drei Eingangswerte in1, in2 und in3 verarbeitet werden und einen Ausgangswert out liefern. Erst mit der expandierten Darstellung des Aktors wird dann auch die konkrete Berechnung sichtbar.



(a) Datenflussaktor kollabiert **(b)** Datenflussaktor expandiert

Abbildung 4.14. Darstellung eines Datenflussaktors mit unterschiedlicher Detailliertheit

4.4. Transformation von Datenfluss in valide SCCharts

Für die Übersetzung von einem SCChart in ausführbaren C-Code sind eine Kette von Modelltransformationen auszuführen. Die Aufteilung in mehrere einzelne Modelltransformationen statt einer einzigen vollständigen Transformation hat verschiedene Vorteile. Unter anderem sind durch diese einzelnen Schritte der Modelltransformationen die Fehlerbehebung oder auch die Validierung von Modellen deutlich einfacher. Dabei wird der Single-Pass Language-Driven Incremental Compilation (SLIC) Ansatz, wie von Motika et al. beschrieben [MSvH14], verwendet. Das bedeutet, dass jede Modelltransformation beim Durchlaufen der vollständigen Übersetzung nur ein einziges Mal ausgeführt werden muss.

4.4.1. Konzept der Datenflusstransformation

Wie in Abschnitt 3.2.1 beschrieben, besitzen SCCharts eine kleine Menge von Core-Features und zusätzliche Extended-Features. In meinem Lösungsansatz werden alle neuen, für den Datenfluss benötigten, Komponenten als neue Extended-Features hinzugefügt. Der Vorteil von diesem Ansatz ist, dass nur eine Transformation zu implementieren ist, die diese neuen Extended-Features in bereits bestehende Extended- oder Core-Features transformiert. Eine weitere Anpassung der anderen Modelltransformationen ist somit nicht notwendig, da alle

Datenflussumgebungen frühzeitig durch äquivalente klassische SCCharts-Regionen substituiert werden.

Damit der SLIC Ansatz weiterhin bestehen bleibt, muss die Transformation von Datenfluss in der Modelltransformationskette so platziert werden, dass keine Zyklen auftreten können. Dieser gesuchte Platz ist ganz am Anfang der Transformationskette. Der Grund dafür liegt in der Verwendungsmöglichkeit von referenzierten SCCharts. Die Referenz muss als erste Transformation ausgeführt werden da zu diesem Zeitpunkt noch nicht ersichtlich ist, welche Elemente und somit benötigten Transformationen in diesem referenzierten SCChart vorhanden und auszuführen sind.

Betrachtet man die erste Gruppe aus der vollständigen Transformationskette (siehe dazu Abbildung 3.6) im Detail, erhält man einen Überblick aller Modelltransformationen aus der Gruppe der Extended-SCCharts. In Abbildung 4.15 ist diese Gruppe der Modelltransformationen mit Fokus der Untergruppe SCCharts Expansions dargestellt. Wie zu erkennen, wird die Referenztransformation als erste Transformation ausgeführt. Für meinen Lösungsansatz wurde diese Referenztransformation so angepasst, dass sie nicht nur die bisherige Verwendung von referenzierten SCCharts auflöst, sondern auch alle Datenflussumgebungen gleichzeitig durch äquivalente SCCharts-Regionen ersetzt.

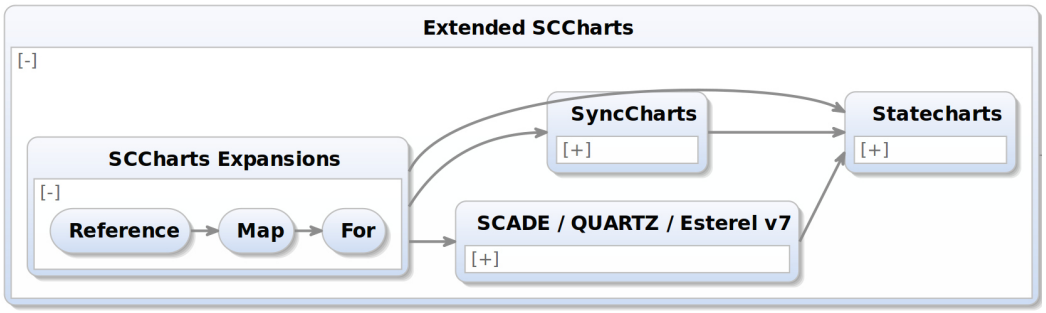


Abbildung 4.15. Übersicht der Modelltransformationen der Extended SCCharts

Für die Substitution der Datenflussumgebungen sind verschiedene Teilschritte notwendig. Das sind zum einen die Auflösung der Referenzknoten und der aufrufenden Knoten. Und des Weiteren die Transformation aller Ausdrücke der direkten Datenflussmodellierung. Da referenzierte SCCharts wiederum Referenzen enthalten können, ruft sich diese Transformation rekursiv auf. Die komplette Transformation läuft somit hierarchisch von außen nach innen ab.

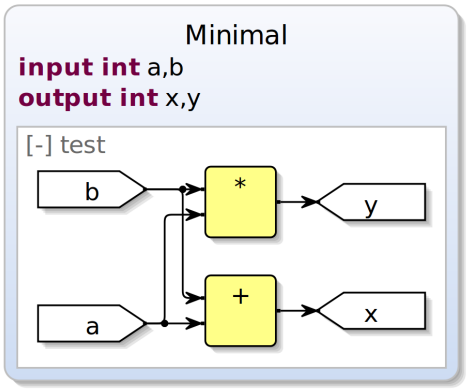
Eine Unterteilung der notwendigen Transformationsschritte in eine Modelltransformation für Datenfluss und eine für Referenzen ist an dieser Stelle nicht möglich. Das heißt, diese Transformation ist eine atomare Transformation. Der Grund dafür liegt in der beliebigen Verschachtelung von Daten- und Kontrollfluss. Ein aufrufender Knoten kann beispielsweise auf einen Datenflussaktor

verweisen, auf dessen oberster Ebene Kontrollfluss modelliert ist. In einem Zustand dieser Kontrollflusskomponente könnte erneut ein Referenzknoten in einer Datenflussumgebung definiert sein. Würde die Referenztransformation einzeln zuerst durchlaufen werden, so müsste man bei der späteren Transformation des aufrufenden Knoten erneut zurückwechseln zur Referenztransformation und somit das Kriterium des SLIC-Ansatzes verletzen. Analog dazu würde bei umgekehrter Ausführungsreihenfolge das Kriterium ebenfalls verletzt werden, wenn in dem referenzierten SCChart ein aufrufender Knoten enthalten wäre. Der rekursive Aufruf einer Modelltransformation verletzt den SLIC Ansatz hingegen nicht.

4.4.2. Erweiterung der Referenztransformation

Es folgt nun eine Vorstellung der konkreten Ideen zur Erweiterung der Referenztransformation. Eine Datenflussgleichung beschreibt die Wertzuweisung einer Variablen. Mit meinem Konzept werden diese Variablenzuweisungen bei der Transformation als Effekte einer Transition zwischen zwei Zuständen generiert. Das bedeutet, dass zunächst eine neue (klassische) Region erstellt wird und diese Region einen Start- und einen Endzustand bekommt. Es wird eine Transition erstellt, die diese beiden Zustände verbindet. Jede Datenflussgleichung der Form $x = e$ wird als eine Zuweisung der Liste von Effekten dieser Transition hinzugefügt. Im Allgemeinen erwartet man, dass der Datenfluss instantan, auch *immediate* genannt, abläuft. Deshalb wird diese Transition auch instantan dargestellt. Die Transition bekommt keinen *Trigger*, damit sie auf jeden Fall durchlaufen wird. Alle Effekte werden ausgelöst sobald die Transition ausgeführt wird. Da eine neue Region mit der transformierten Repräsentation der Datenflussgleichungen erstellt wird, wird nach der Transformation die alte Datenflussumgebung gelöscht. In Abbildung 4.16 ist ein kleines Beispiel mit zwei Datenflussgleichungen zu sehen. Links oben die graphische Repräsentation, rechts oben die dazugehörige textuelle Beschreibung des Modells. Abbildungsbereich 4.16c zeigt das Modell nach der Ausführung der Referenztransformation. Wie zu erkennen ist, sind die beiden beschriebenen Gleichungen nun als Effekte einer instantanen Transition vorhanden.

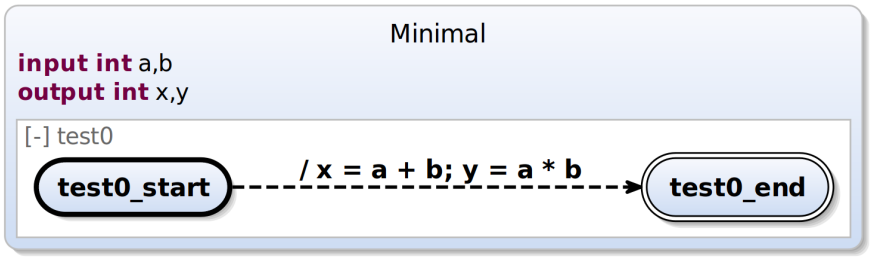
Neben der instantanen Berechnung von Datenfluss wird im Allgemeinen auch davon ausgegangen, dass diese Berechnung in jedem Tick durchgeführt wird. Um dies zu gewährleisten muss die Transformation leicht abgeändert werden. In der neuen Region gibt es nach der Transformation keinen Endzustand mehr. Aus diesem zweiten (normalen) Zustand führt dafür eine weitere Transition zurück zum Startzustand. Diese zweite Transition ist nicht immediate, so dass sie mit einem Tick Verzögerung ausgeführt wird. Dadurch wird gewährleistet, dass der Datenfluss in jedem Tick ausgeführt wird. Zur Verdeutlichung ist in Abbildung 4.16d das eben vorgestellte Beispiel nach der Ausführung dieser zweiten Variante der Referenztransformation dargestellt.



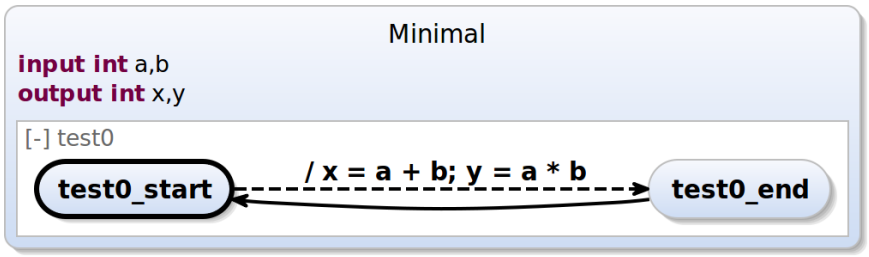
(a) Visualisierung mit KlighD

```
1 scchart Minimal {
2   input int a, b;
3   output int x, y;
4
5   dataflow test:
6     x = a + b;
7     y = a * b;
8 }
```

(b) Modellierung in SCT



(c) Transformation mit Start- und Endzustand

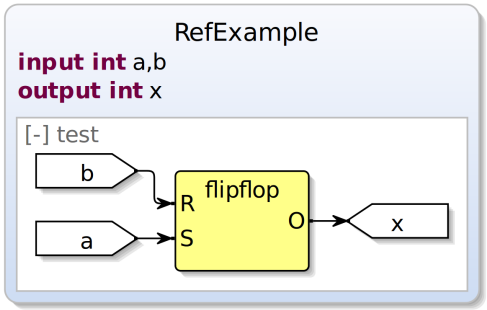


(d) Transformation ohne Endzustand

Abbildung 4.16. Darstellung der Referenztransformation an einem kleinen Beispiel mit zwei Datenflussgleichungen

Generell bleibt dem Entwickler immer die Möglichkeit gegeben ein Konstrukt zur Berechnung von Datenfluss in jedem Tick auszuführen selbst zu modellieren. Unabhängig von der Implementierung der Referenztransformation kann einem Zustand, welcher Datenfluss enthält, eine Selbsttransition hinzugefügt werden. Diese darf nicht instantan sein. Damit wird gewährleistet, dass die Datenflussberechnungen in jedem Tick ausgeführt werden.

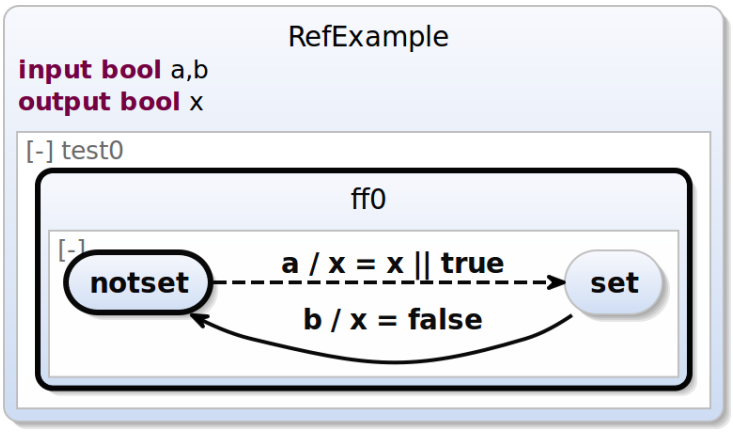
Als nächstes betrachte ich die Transformation der verschiedenen Knotentypen. Ein referenzierender Knoten verweist auf ein separates SCChart. Jedes KIELER SCChart ist als Wurzelelement eine Instanz vom Typ State. Bei der Transformation wird eine neue Region mit einem neuen Zustand erzeugt in den dann der vollständige Inhalt des referenzierten SCChart hineinkopiert wird. Nachdem der Inhalt kopiert wurde müssen noch alle Variablen in diesem referenzierten SCChart ersetzt werden, so dass sie mit den tatsächlich verbundenen Ein- und Ausgangsvariablen des referenzierenden Knoten übereinstimmen. Zur Verdeutlichung ist in Abbildung 4.17 ein SCChart mit einer Datenflussumgebung zu sehen, in der das Flipflop



(a) Visualisierung mit KlighD

```
1 scchart RefExample {
2   input int a, b;
3   output int x;
4
5   dataflow test:
6     ff = ref flipflop(a, b);
7     x = ff.O;
8 }
```

(b) Modellierung in SCT

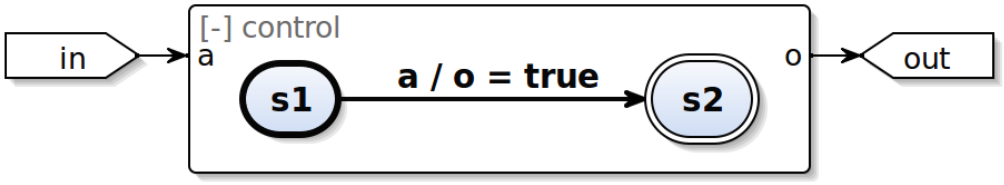


(c) Visualisierung mit KlighD (transformiert)

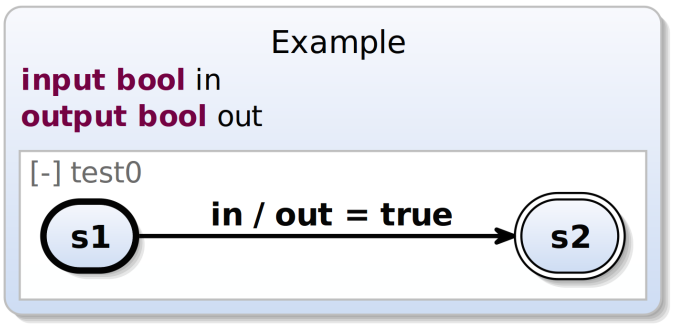
Abbildung 4.17. Darstellung der Referenztransformation am Beispiel eines referenzierenden Knotens

aus Abbildung 4.2 referenziert wird. Im oberen Bereich ist das graphische und das textuelle Modell zu sehen. Im unteren Bereich ist das transformierte SCChart mit dem referenzierten Flipflop dargestellt.

Ein definierender Knoten wird nicht direkt transformiert. Bereits bei der Visualisierung wird das Verhalten eines definierenden Knotens erst dargestellt, wenn ein aufrufender Knoten auf ihn zugreift. Bei der Transformation verhält es sich genauso. Es werden alle aufrufenden Knoten transformiert, da diese ja auf einen definierenden Knoten verweisen. Enthält der definierende Knoten Datenflussgleichungen, so wird wie oben bereits erläutert, für jede Gleichung ein Effekt einer instantanen Transition generiert. Enthält ein definierender Knoten Kontrollfluss, so werden keine neuen Zustände erzeugt. Der modellierte Kontrollfluss wird in eine neue (klassische) Region kopiert. Anschließend müssen wie bei den referenzierenden Knoten alle Variablen in den Datenflussgleichungen durch die tatsächlich beim Aufruf verwendeten Variablen ersetzt werden. In Abbildung 4.18 ist das transformierte Beispiel aus Abbildung 4.11 zu sehen.



(a) Beispielmodell aus Abbildung 4.11



(b) Beispielmodell nach der Transformation

Abbildung 4.18. Transformation eines Knotenaufrufs, der Kontrollfluss enthält

Die Referenztransformation ruft sich nach dem Durchlaufen einer Datenflussumgebung rekursiv selbst auf. Dies ist notwendig, da in einem referenzierten SCChart sich wiederum Daten- oder Kontrollfluss befinden kann. Des Weiteren kann in einem definierenden Knoten, der Kontrollfluss enthält, auch wiederum eine Datenflussumgebung in einem der Zustände modelliert sein. Nur mit dem rekursiven Aufruf werden alle Datenflussumgebungen vollständig transformiert.

Kapitel 5

Implementierung

Nach der Vorstellung des Konzepts im letzten Kapitel folgt nun eine Beschreibung der Implementierung von Datenfluss in SCCharts. In Unterkapitel 5.1 werden einige weitere konkrete Änderungen am Metamodell der SCCharts und der dazugehörigen Grammatik beschrieben. Unterkapitel 5.2 beschreibt die Anpassung der Diagrammsynthese. Abschließend wird in Unterkapitel 5.3 auf die Erweiterung der Referenztransformation eingegangen.

5.1. Erweiterung von Metamodell und Grammatik

Der erste Schritt zur Implementierung von Datenfluss in SCCharts ist die Einarbeitung der Erweiterungen in das Metamodell der SCCharts. Das Konzept und die Verwendung der Datenflusselemente wurde bereits in Unterkapitel 4.2 beschrieben. Der folgende Abschnitt geht noch auf einige Besonderheiten der Implementierung ein.

5.1.1. Implementierung der Datenflusskomponenten

Datenflussumgebung Der Datenfluss wird in einer neuen nebenläufigen Umgebung modelliert. Diese neue Klasse im Metamodell der SCCharts heißt `Dataflow` und leitet sich von der Klasse `Concurrency` ab. Im Gegensatz zum bestehenden Metamodell der SCCharts (siehe Abbildung 3.3) hat ein Zustand (`State`) nicht mehr direkt eine Referenz auf die Klasse `Region`, sondern auf die Klasse `Concurrency`. Somit können Daten- und Kontrollfluss beliebig nebenläufig modelliert werden.

Die Klasse der Datenflussumgebung besitzt zwei Referenzen, wie in Abbildung 4.5 dargestellt. Die erste Referenz ist auf die Klasse `Equation` und kann beliebig viele Elemente enthalten. Mit den Instanzen dieser Klasse lassen sich die Datenflussgleichungen zur direkten Modellierung in einer Umgebung beschreiben. Die zweite Referenz kann ebenfalls beliebig viele Elemente enthalten und verweist auf die Klasse `Node`. Mit den Elementen dieser Klasse lassen sich in einer Datenflussumgebung die verschiedenen Knotentypen modellieren.

Datenflussgleichungen Die Modellierung von Datenflussgleichungen wird wie in Abschnitt 4.2.2 erläutert als Gleichung der Form:

$$< valuedObject > = < expression >$$

interpretiert. Für den Fall, dass auf den Ausgangswert eines Knotens zugegriffen wird, muss die optionale Referenz auf die Klasse Node verwendet werden. Dies erfolgt in der Form:

$$< valuedObject > = < node > . < valuedObjectReference >$$

Der angegebene Knoten muss dabei ein in der Datenflussumgebung vorhandener referenzierender oder aufrufender Knoten sein. Der Zugriff auf die Ausgangswerte dieses Knotens wird in Abschnitt 5.1.2 näher erläutert.

Knoten Die Klasse Node dient zur Beschreibung der Knoten und besitzt zwei Attribute. Einmal das Attribut id für die Vergabe eines Bezeichners und des Weiteren das Attribut label, um einen Namen anzugeben. Von dieser Klasse leiten sich die drei Unterklassen DefineNode, CallNode und ReferenceNode ab, welche die drei in Unterkapitel 4.2.3 vorgestellten Knotentypen beschreiben. Ein detaillierter Ausschnitt des Metamodells zur Einbindung der Klasse Node ist in Abbildung 4.9 dargestellt.

Ein definierender Knoten kann entweder Datenfluss oder Kontrollfluss beschreiben, aber nicht beides zusammen. Listing 5.1 zeigt dazu noch einmal einen Ausschnitt aus der Grammatikregel zur Beschreibung eines definierenden Knotens. Der gegenseitige Ausschluß beider Modellierungskonzepote wird durch die Veroderung (der senkrechte Strich in Zeile vier) festgelegt.

```
1 (
2   ((valuedObjects += [kexpressions::ValuedObject])
3                  '=' (expressions += Expression) ';')*
4   |
5   (states += State)*
6 )
```

Listing 5.1. Grammatikregel eines definierenden Knotens (Ausschnitt)

5.1.2. Der ScopeProvider

Zur Erleichterung der Modellierung wird ein sogenannter ScopeProvider eingebunden. Dieser ScopeProvider wird an zwei Stellen eingesetzt. Die erste Anwendung dieser Hilfsfunktion liefert bei den referenzierenden Knoten eine Auswahl an Zuständen auf welche referenziert werden kann (also separate SCCharts). Der Codeausschnitt zur Generierung dieser Anzeige ist in Listing 5.2 dargestellt.

```
1  public def IScope scope_referenceNode_referencedScope(EObject context,
2                                                        EReference reference) {
3    val superScope = super.getScope(context.eContainer, reference)
4    val res = context.eResource
5    if (res != null) {
6      val resSet = res.resourceSet
7      if (resSet != null) {
8        val rIterable = <Scope>newArrayList
9        for (r : resSet.resources) {
10         val contentList = r.contents.filter(e|e instanceof State).toList
11         if (!contentList.nullOrEmpty) {
12           for (content : contentList) {
13             rIterable += content as Scope
14           }
15         }
16       }
17       return Scopes.scopeFor(rIterable, nameProvider, superScope);
18     }
19   }
20   return IScope.NULLSCOPE;
21 }
```

Listing 5.2. Codeausschnitt des ScopeProviders bei referenzierenden Knoten

Damit die Referenzierung auf externe Dateien funktionieren kann muss die Funktion *Xtext Nature* im aktuellen Projekt der laufenden KIELER Instanz aktiviert sein. Die im Listing dargestellte Methode bekommt zunächst nur ein EObject und eine EReference mit übergeben, welche den aktuellen Kontext repräsentieren. Danach wird auf die Resource an sich (das aktuelle SCChart) und das diese beinhaltende Resourcenset zugegriffen. Dieses Resourcenset wird anschließend nach weiteren Resourcen (andere SCCharts im gleichen Projekt) durchsucht und es werden alle Elemente herausgefiltert, die vom Typ State sind. Das sind die gewünschten SCCharts, welche referenziert werden können. Abschließend wird ein Scope für die Liste dieser SCCharts zurückgegeben.

Der zweite Anwendungsfall des ScopeProviders ist eine Hilfestellung um die Ausgangswerte von referenzierenden und aufrufenden Knoten zugreifen zu können. In Listing 5.3 ist der erste Codeausschnitt dazu zu sehen. Wird bei der Modellierung eine Variablenreferenz erwartet, überpüft diese Methode ob der aktuelle Kontext eine Instanz der Klasse Equation ist. Ist dies der Fall, so wird die Methode zur Generierung der Auswahl an möglichen Ausgangsvariablen, dargestellt in Listing 5.4, aufgerufen. Diese Methode greift auf die Referenz node der aktuellen Gleichung (equation) des übergebenen Kontextes zu. Der Wert dieser Referenz wird in der Variablen obj in Zeile vier des Listings gespeichert. Es folgt eine Überprüfung ob die Knotenreferenz vom Typ eines referenzierenden Knoten

```
public def IScope scope_ValuedObjectReference_valuedObject(EObject context,
                                                           EReference reference) {
  if (context instanceof Equation) {
    return equation_ValuedObjectReferenceScope(context, reference)
  }
  return null;
}
```

Listing 5.3. Codeausschnitt des ScopeProviders für eine Variablenreferenz

```
public def IScope equation_ValuedObjectReferenceScope(EObject context,
                                                      EReference reference) {
  var EObject theContext = context;
  val obj = theContext.eGet(pack.equation_Node, true)
  if (!isProxy(obj)) {
    // case of RefNode: get outputs of referencedScope (scchart)
    if (obj instanceof ReferenceNode) {
      var refNode = obj as ReferenceNode
      val volterable = <ValuedObject>newArrayList
      refNode.referencedScope.declarations.forEach[
        if (it.output) {
          volterable += valuedObjects
        }
      ]
      return Scopes.scopeFor(volterable)
    }
    // case of CallNode: get outputs of referenced DefineNode
    else if (obj instanceof CallNode) {
      var refNode = obj as CallNode
      val volterable = <ValuedObject>newArrayList
      refNode.callReference.outputs.forEach[
        volterable += valuedObjects
      ]
      return Scopes.scopeFor(volterable)
    }
  } else {
    return IScope.NULLSCOPE
  }
}
```

Listing 5.4. Codeausschnitt des ScopeProviders für Ausgangsvariablen

ist. Trifft dies zu, so wird eine Liste der Ausgangsvariablen des referenzierten SCChart erstellt. Dies erfolgt über die Deklarationen des separaten SCChart. Die Liste dieser Ausgangsvaribalen wird als `Scope` zurückgegeben. Ab Zeile 17 ist der Fall für einen aufrufenden statt referenzierenden Knoten zu sehen. Die Generierung der Liste der Ausgangsvaribalen ist etwas einfacher, da ein definierender Knoten eine einzelne Liste von Deklarationen, die ausschließlich Ausgangswerte repräsentieren, besitzt.

5.2. Anpassung der Diagrammsynthese

Das folgende Unterkapitel beschreibt die Diagrammsynthese und Visualisierung von Datenfluss in SCCharts mittels KLighD. Dazu wird die bereits bestehende Synthese erweitert. Der Ablauf der Diagrammsynthese gestaltet sich wie folgt. Es werden alle Datenflussumgebungen in einem SCChart übersetzt. Innerhalb jeder Umgebung werden alle Knoten synthetisiert. Handelt es sich dabei um einen aufrufenden Knoten, so wird der darin modellierte Daten- beziehungsweise Kontrollfluss ebenfalls übersetzt. Abschließend erfolgt noch die Synthese aller direkt modellierten Datenflussgleichungen innerhalb einer Umgebung.

Eine Datenflussumgebung ist zunächst wie die bereits existierende nebenläufige `Region` aufgebaut. Innerhalb dieser Umgebung werden dann alle dazugehörigen Elemente der Datenflussmnodellierung dargestellt. Der Anfangszustand der Darstellung einer Umgebung ist der expandierte Modus. Möchte man aus Gründen der Übersichtlichkeit einzelne Umgebungen kollabieren, muss man dafür auf das „-“ in der oberen, linken Ecke dieser Umgebung klicken.

Die einzelnen Komponenten innerhalb einer Datenflussumgebung werden als grafische Elemente platziert. In Abbildung 5.1 sind folgende drei Standardelemente dargestellt. Ein Eingangswert hat die Form eines nach rechts zeigenden Pfeils. Analog dazu zeigt die Grafik eines Ausgangswertes nach links. Diese Repräsentation ist so gewählt, da der Lesefluss von links nach rechts verläuft. Generell bietet KLighD als Diagrammoption auch an, die dargestellten Elemente in einer anderen Leserichtung anzuordnen, zum Beispiel von oben nach unten. Damit der Lesefluss dann weiterhin sinnvoll gegeben ist, müsste man die Piktogramme der Elemente (in diesem Fall das Symbol eines Pfeils) und die entsprechenden Ports

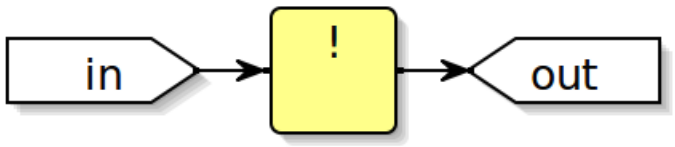


Abbildung 5.1. Darstellung von Standardkomponenten von Datenfluss in SCCharts

rotieren. In der vorliegenden Implementierung ist dies noch nicht umgesetzt. Die einzelnen (Teil-)Operationen einer Zuweisung werden als Quadrate mit abgerundeten Ecken dargestellt. Der entsprechende Operator wird als Textlabel innerhalb dieses Quaders am oberen Rand platziert. In Abbildung 5.1 ist dies zum Beispiel der Not-Operator. Die Platzierung aller Elemente in KLighD erfolgt mittels des in KIELER enthaltenen automatischen Layoutalgorithmus KLay Layered. Die verwendete Knotenplatzierungsstrategie ist der Brandes-Köpf-Algorithmus [BK02]. Dieser Algorithmus platziert die im Modell enthaltenen Elemente derart in Blöcken, dass möglichst gerade Kanten zwischen den Elementen generiert werden können.

Eine Kante in einem Datenflussdiagramm stellt den Fluss des Datenaustauschs zwischen verschiedenen Modellkomponenten dar. Zugunsten der Lesbarkeit erfolgt das Kantenrouting orthogonal und die Datenflussrichtung wird durch einen kleinen Pfeil am Ende jeder Kante dargestellt, wie es ebenfalls in Abbildung 5.1 zu sehen ist.

Handelt es sich bei der Diagrammsynthese eines Knotens um einen aufrufenden Knoten, so werden alle Elemente des definierenden Knoten als Kindelemente innerhalb des aufrufenden Knoten platziert. Dazu wird wie in Listing 5.5 in Zeile eins zu sehen, zunächst überprüft, ob der Knoten überhaupt auf einen definierenden Knoten verweist. Die Synthese der Kindelemente erfolgt über alle Variablenzuweisungen des definierenden Knotens, so dass alle Datenflüsse visualisiert werden. Dies ist im Listing 5.5 in den Zeilen zwei bis sieben zu sehen. Für

```
1 if (call.callReference instanceof DefineNode) {
2   val ref = call.callReference as DefineNode
3   // add ChildNodes for called Reference
4   for (expr : ref.expressions) {
5     val index = ref.expressions.indexOf(expr)
6     nNode.children +=  expr.translate(index, call)
7   }
```

Listing 5.5. Codeausschnitt der Synthese eines aufrufenden Knoten

den Fall, dass in dem aufrufenden Knoten statt des Datenflusses Kontrollfluss beschrieben wird, wird die bereits vorhandene Methode zur Synthese von Zuständen verwendet. Der entsprechende Codeausschnitt ist in Listing 5.6 zu sehen. Dabei wird vorher eine zusätzliche Methode delegate aufgerufen, siehe Zeile zehn. Diese Methode ist notwendig, da sie gewährleistet, dass bei einem mehrfachen Aufruf von einem definierenden Knoten mit Kontrollfluss auch in allen Fällen alle Elemente angezeigt werden. Ansonsten würde die Synthese nur für den ersten Aufruf erfolgen und für alle weiteren Aufrufe lediglich auf diese Elemente verweisen. Um die Übersichtlichkeit zu wahren wird wie in Unterkapitel 4.3 beschrieben ein aufrufender Knoten zunächst kollabiert dargestellt.

```
1   // translate states if control flow is defined instead of dataflow
2   if (!ref.states.nullOrEmpty) {
3     val synthesis = delegate.get()
4     synthesis.use(usedContext)
5     for (s: ref.states) {
6       nNode.children += synthesis.translate(s)
7     }
8   }
9 }
```

Listing 5.6. Codeausschnitt der Synthese von Zuständen eines aufrufenden Knoten

Bei der Visualisierung ist darauf zu achten, ob die Anzahl der übergebenen Parameter eines aufrufenden Knoten mit der Anzahl der Variablendeklarationen von Eingangswerten des definierenden Knoten übereinstimmen. Um auch Zwischenschritte der Visualisierung während der Modellierung darstellen zu können, werden bei zu wenigen übergebenen Parametern die jeweiligen Ports einfach frei gelassen. Für die Weiterverarbeitung der Modelle (beispielsweise Codegenerierung) muss man sich überlegen, wie mit freien Ports umgegangen werden soll. Mein Vorschlag ist einen voreingestellten Standardwert entsprechend des Variablentyps dann einzusetzen. Bei der Übergabe zu vieler Parameter würde es ohne eine Überprüfung an dieser Stelle zu einer fehlerhaften Visualisierung kommen. In Listing 5.7 zeigt Zeile sieben diese Überprüfung in Form einer Fallunterscheidung. Hat die Anzahl der synthetisierten Parameter die ursprünglich definierte Anzahl bereits überschritten, so wird keine Synthese weiterer Parameter durchgeführt. Diese Fallunterscheidung bietet sich auch bei der Weiterverarbeitung der Modelle an, damit es zu keinem fehlerhaften Verhalten bei den Modelltransformationen kommt.

```
1  val callRef = call.callReference as DefineNode
2  callRef.inputs.forEach[
3    refInputs += valuedObjects
4  ]
5  val refInputSize = refInputs.size
6  call.parameters.forEach[ p|
7    if (call.parameters.indexOf(p) < refInputSize) {
8      // add child nodes and edges if not already craeted
9      // only if the index of the parameter
10     // is less then the number of inputs of the define node
11   }
12 }
```

Listing 5.7. Codeausschnitt zur Überprüfung zu vieler Parameter

Als letzter Punkt im Verlauf der Diagrammsynthese werden noch alle Datenflussgleichungen der direkten Modellierung, ohne Einbettung in einem Knoten, übersetzt. Ähnlich wie bei den aufrufenden Knoten werden dazu alle (Teil-) Ausdrücke der Variablenzuweisungen ausgewertet. Bei dieser Auswertung wird eine Fallunterscheidung (switch) anhand des Typs des Ausdrucks (expr) vorgenommen. Diese Fallunterscheidung ist in Listing 5.8 schematisch dargestellt. Der

```
1 switch(expr) {
2   OperatorExpression: { ... }
3   IntValue: { ... }
4   FloatValue: { ... }
5   BoolValue: { ... }
6   ValuedObjectReference: { ... }
7 }
```

Listing 5.8. Codeausschnitt zur Fallunterscheidung verschiedener Ausdrücke

Ausdrucktyp OperatorExpression beschreibt eine beliebig komplexe Variablenzuweisung verschiedener (Teil-)Operationen. Für die Verwendung von konstanten Werten gibt es Ausdruckstypen entsprechend ihres Variablentyps: IntValue für ganzzahlige Werte, FloatValue für Fließkommazahlen und BoolValue für boolesche Werte. Der Typ ValuedObjectReference beschreibt die direkte Verwendung einer Variablen, welche ein ValuedObject ist.

5.3. Anpassung der Referenztransformation

Dieses Unterkapitel beschreibt den dritten Teil meiner Implementierung, die Anpassung der Referenztransformation. Wie in Unterkapitel 4.4 erläutert, besteht die Datenflussmodellierung aus neuen Extended-Features, welche in valide Core-SCCharts übersetzt werden müssen. Dazu ist eine Anpassung der Referenztransformation notwendig. Die Transformation von Datenfluss wird noch vor der bereits existierenden Funktionalität der Referenztransformation ausgeführt. Listing 5.9 zeigt den entsprechenden Programmausschnitt. Die Methode transformDataflows, dargestellt in Zeile 14 des Listings 5.9, übersetzt alle Datenflussumgebungen eines Zustandes. Dabei werden zuerst die referenzierenden Knoten transformiert, siehe Zeile 21, dann die aufrufenden Knoten in Zeile 23 und abschließend dann alle Datenflussgleichungen der jeweiligen Umgebung. Die detaillierten Abläufe dieser Transformation der einzelnen Elemente wird in den kommenden Abschnitten näher erläutert.

```
1  def State transform(State rootState) {
2    val targetRootState = rootState.fixAllPriorities;
3    // Transform dataflows first
4    targetRootState.transformDataflows
5    // Traverse all referenced states
6    targetRootState.allContainedStates.filter[ referencedState ]
7                                      .toList.immutableCopy.forEach[
8      transformReference(targetRootState)
9    ]
10
11   targetRootState;
12 }
13
14 def transformDataflows(State state) {
15   val dataflows = <Dataflow> newHashSet
16   // traverse all dataflows
17   state.getAllContainedStates.forEach[
18     dataflows += concurrencies.filter(typeof(Dataflow))
19   ]
20   for(dataflow : dataflows.immutableCopy) {
21     for(refNode: dataflow.nodes.filter(typeof(ReferenceNode))) { ... }
22
23     for(callNode: dataflow.nodes.filter(typeof(CallNode))) { ... }
24
25     for (eq: dataflow.equations) { ... }
26   }
27 }
```

Listing 5.9. Codeausschnitt zur Transformation von Datenfluss

5.3.1. Transformation von referenzierenden Knoten

Der erste Teil der Datenflusstransformation übersetzt alle referenzierenden Knoten. Listing 5.10 zeigt einen ersten Programmausschnitt zur Transformation dieses Knotentyps. Für jeden dieser Knoten innerhalb einer Datenflussumgebung wird eine neue Region im Elternzustand des Knotens angelegt, siehe Zeile zwei. Innerhalb dieser Region wird dann noch ein neuer Zustand generiert, wie in Zeile vier zu sehen ist. In Zeile zehn wird dann das Modell des referenzierten SCChart in diesen neuen Zustand hineinkopiert. Nachdem der Inhalt kopiert wurde, müssen anschließend noch alle Eingangs- und Ausgangsvariablen des referenzierten SCChart durch die tatsächlich verwendeten Variablen ersetzt werden. Diese Anpassung der Eingangswerte ist in Listing 5.11 zu sehen. Dazu wird zunächst eine Auflistung aller im referenzierten SCChart enthaltenen Eingangsvariablen (refedInputs) erstellt. Danach werden die Parameter des referenzierenden Knoten durchlaufen. Jede Variable (valuedObject) wird nun als neuer Wert (newBinding.actual) anstelle des alten Wertes (newBinding.formal) geschrieben. Anschließend werden auf ähnliche

```
for(refNode: dataflow.nodes.filter(typeof(ReferenceNode))) {
  val rRegion = parentState.createRegion("_" + dataflow.id + regionCounter)
  rRegion.label = dataflow.label + regionCounter
  val newState = rRegion.createState("_" + refNode.ID + idCounter)
  newState.label = if (refNode.label != null) {refNode.label + idCounter}
                   else {refNode.id + idCounter}
  regionCounter = regionCounter + 1
  idCounter = idCounter + 1
  newState.setInitial
  newState.referencedScope = refNode.referencedScope
  nodeMapping.put(refNode, newState)
```

Listing 5.10. Codeausschnitt zur Transformation von referenzierenden Knoten

```
  // bind inputs
  var exprCounter = 0
  val refedInputs = <ValuedObject>newArrayList
  refNode.referencedScope.declarations.filter[it.input].forEach[
    refedInputs += valuedObjects
  ]
  for (expr: refNode.parameters) {
    val newBinding = SCChartsFactory.eINSTANCE.createBinding
    newBinding.actual = (expr as ValuedObjectReference).valuedObject
    newBinding.formal = refedInputs.get(exprCounter)
    val rState = nodeMapping.get(refNode)
    rState.bindings += newBinding
    exprCounter = exprCounter + 1
  }
```

Listing 5.11. Codeausschnitt zur Transformation von referenzierenden Knoten, Teil 2

Weise auch die Ausgangswerte angepasst. Listing 5.12 zeigt den entsprechenden Programmausschnitt. Da die Ausgangswerte der referenzierenden Knoten beliebig oft verwendet werden können, wird diesmal die Liste der Datenflussgleichungen durchlaufen. Zunächst wird dabei überprüft, ob die Gleichung auf einen Knoten zugreift und wenn ja, ob dieser Knoten derjenige ist, der gerade transformiert wird. Ist der Ausdruck eine Variablenreferenz (ValuedObjectReference) des referenzierenden Knotens, dann wird diese durch den neuen Wert ersetzt.

```
1  // bind outputs
2  val refedOutputs = <ValuedObject>newArrayList
3    refNode.referencedScope.declarations.filter[it.output].forEach[
4    refedOutputs += valuedObjects
5  ]
6  for (eq: dataflow.equations) {
7    if (eq.node != null) {
8      if (eq.node.equals(refNode)) {
9        val refedVo = (eq.expression as ValuedObjectReference).valuedObject
10       if (refedOutputs.contains(refedVo)) {
11         val newBinding = SCChartsFactory.eINSTANCE.createBinding
12         newBinding.actual = eq.valuedObject
13         newBinding.formal = refedVo
14         val rState = nodeMapping.get(refNode)
15         rState.bindings += newBinding
16       }
17     }
18   }
19 }
```

Listing 5.12. Codeausschnitt zur Transformation von referenzierenden Knoten, Teil 3

5.3.2. Transformation von aufrufenden Knoten

Nach der Transformation der referenzierenden Knoten werden die aufrufenden Knoten transformiert. Alle Knoten vom Typ CallNode werden dafür transformiert, wie in Zeile eins des Listings 5.13 zu sehen ist. Die Listen refedInputs und refedOutputs stellen alle in Deklarationen eingebettete Eingangs- und Ausgangsvariablen in jeweils einer einzelnen Liste zusammen, damit ein Abgleich später einfacher ist. Bei den aufrufenden Knoten werden zwei Fälle unterschieden. Das hängt von dem Inhalt des definierenden Knoten ab, auf den verwiesen wird. Da nur entweder Datenflussgleichungen oder Zustände innerhalb des Knotens

```
1  for(callNode: dataflow.nodes.filter(typeof(CallNode))) {
2    val defNode = callNode.callReference
3    val refedInputs = <ValuedObject>newArrayList
4    defNode.inputs.forEach[
5      refedInputs += valuedObjects
6    ]
7    val refedOutputs = <ValuedObject>newArrayList
8    defNode.outputs.forEach[
9      refedOutputs += valuedObjects
10   ]
```

Listing 5.13. Codeausschnitt zur Transformation von aufrufenden Knoten

modelliert werden können, reicht eine einfache Überprüfung, ob die Anzahl der Zustände null ist.

Zunächst betrachte ich den Fall, dass Datenflussgleichungen in dem definierenden Knoten modelliert sind. Der Programmausschnitt dazu ist in Listing 5.14 dargestellt. Als erstes wird wieder im Elternzustand eine neue Region angelegt, siehe Zeile drei. Diese bekommt einen Startzustand und einen Endzustand, was in Zeile sechs bis zwölf zu sehen ist. Eine Transition zwischen diesen beiden Zuständen wird ebenfalls erzeugt. Wie in Unterkapitel 4.4 bereits erläutert, muss diese Transition *immediate* sein, also keinen Tick verbrauchen. Dies ist in Listing 5.14 in den Zeilen 14 und 15 zu sehen.

```
1  if (defNode.states.nullOrEmpty) {
2    // case: dataflow is modelled inside define node
3    val newRegion = parentState.createRegion("_" + dataflow.id + regionCounter)
4    newRegion.label = dataflow.label + regionCounter
5
6    val newState = newRegion.createState("_" + dataflow.ID + idCounter)
7    newState.label = dataflow.label + idCounter + "_start"
8    newState.setInitial
9
10   val newState2 = newRegion.createState("_" + dataflow.ID + idCounter)
11   newState2.label = dataflow.label + idCounter + "_end"
12   newState2.setFinal
13
14   val transition = createNewTransition(newState, newState2)
15   transition.setImmediate
16
17   var eqCounter = 0
18   for (vo: defNode.valuedObjects) {
19     val newAssignment = createNewAssignment(vo,
20                           defNode.expressions.get(eqCounter).copy)
21
22     transition.effects += newAssignment
23     eqCounter = eqCounter + 1
24
25     assignmentMapping.put(callNode.id + "." + vo.name, newAssignment)
26   }
27
28   // replace inputs
29   var exprCounter = 0
30   for (p: callNode.parameters) {
31     val in = (p as ValuedObjectReference).valuedObject
32     newState.replaceAllOccurrences(refedInputs.get(exprCounter), in)
33     exprCounter = exprCounter + 1
34   }
```

Listing 5.14. Codeausschnitt zur Transformation von aufrufenden Knoten, Teil 2

Der nächste Schritt ist dann die Transformation der im definierenden Knoten enthaltenen Datenflussgleichungen. Dies geschieht in Zeile 17 bis 26 des Listings 5.14. Für alle Variablenzuweisungen des definierenden Knotens wird eine neue Instanz der Klasse Assignment angelegt. Die aktuelle Variable und der dazugehörige Ausdruck aus dem Knoten werden dieser neuen Zuweisung übergeben. Diese Zuweisungen werden dann als Effekte der zuvor angelegten Transition hinzugefügt. Wie schon bei den referenzierenden Knoten erklärt, müssen nun alle Eingangsvariablen durch die tatsächlich verwendeten Variablen ersetzt werden. Dies erfolgt in Listing 5.14 in Zeile 29 bis 34.

Die Zuweisung der tatsächlich verwendeten Ausgangsvariablen ist im Gegensatz zu den Eingangsvariablen etwas umfangreicher. Wie in Unterkapitel 4.4 erläutert, soll ein aufrufender Knoten nur einmal transformiert werden, auch wenn mehrere Ausgangsvariablen auf den Ausgangswert des Knotens zugreifen. Das heißt für die Transformation, dass die Berechnung des Ausdrucks nur einmal stattfinden soll. Wenn mehrere Ausgangsvariablen auf einen Ausgangswert des Knotens zugreifen, wird also ein Zwischenschritt benötigt. In Listing 5.15 ist der Codeausschnitt zur Zuweisung der Ausgangsvariablen aufgelistet. Zunächst gilt es, die Datenflussgleichungen herauszufiltern welche überhaupt einen Verweis auf den aktuellen aufrufenden Knoten haben. Danach wird in Zeile neun überprüft, ob dieser Ausgangswert schon einmal vorgekommen ist. Wenn ja, dann wird in Zeile zwölf überprüft, ob die Anzahl der Zustände in der neuen Region immer noch zwei ist. Weniger als zwei Zustände können nicht vorhanden sein, da mindestens die zwei bereits generierten Zuständen enthalten sind. Sind mehr als zwei Zustände vorhanden, wurde bereits ein dritter Zustand als Zwischenschritt erzeugt. Sollte noch kein neuer Zwischenzustand vorhanden sein, so wird in der Region ein neuer Zustand generiert. Der alte Endzustand bekommt nicht mehr das Attribut final, dafür aber der neu generierte Zustand. Zwischen diesen beiden Zuständen wird eine neue instantane Transition erzeugt, siehe Zeile 22 und 23. Jetzt wird noch eine neue Zuweisung als Effekt dieser Transition hinzugefügt. Da die Ausgangsvariable bereits einmal vorgekommen ist, kann der Ausdruck dieser neuen Zuweisung einfach auf die alte Variable referenzieren. Dies geschieht über den Aufruf von existingValuedObject.reference in Zeile 27 des Listings. Ist der aktuell betrachtete Ausgangswert noch nicht vorgekommen, so muss lediglich in der anfangs generierten Zuweisung das valuedObject, also die Variable des definierenden Knotens, durch die aktuelle Ausgangsvariable ersetzt werden. Dies geschieht in dem else-Zweig in Zeile 30 bis 36.

Abschließend, nachdem das Ersetzen der Ausgangsvariablen durchlaufen ist, werden noch die Zähler für die neu anzulegenden Regionen erhöht.

```
1  // replace outputs
2  for (eq: dataflow.equations) {
3    if (eq.node != null) {
4      if (eq.node.equals(callNode)) {
5        val key = callNode.id + "."
6                + (eq.expression as ValuedObjectReference).valuedObject.name
7        val vo = eq.valuedObject
8
9        if (voMapping.containsKey(key)) {
10         val existingValuedObject = voMapping.get(key)
11
12         if (newRegion.states.size == 2) {
13           // create new final state
14           newState2.setFinal(false)
15           newState2.label = dataflow.label + idCounter + "_mid"
16
17           val newFinalState = newRegion.createState("_" + dataflow.id
18                                                     + idCounter)
19           newFinalState.label = dataflow.label + idCounter + "_end"
20           newFinalState.setFinal
21
22           val newTransition = createNewTransition(newState2, newFinalState)
23           newTransition.setImmediate
24         }
25
26         val newAssignment = createNewAssignment(vo,
27                                 existingValuedObject.reference)
28         newState2.outgoingTransitions.get(0).effects += newAssignment
29
30       } else {
31
32         val oldAssignment = assignmentMapping.get(key)
33         oldAssignment.valuedObject = vo
34
35         voMapping.put(key, vo)
36       }
37     }
38   }
39 }
40
41 regionCounter = regionCounter + 1
42 idCounter = idCounter + 1
43
44 // remember CallNode and its initial state
45 nodeMapping.put(callNode, newState)
```

Listing 5.15. Codeausschnitt zur Transformation von aufrufenden Knoten, Teil 3

Als nächstes beschreibe ich den zweiten Fall, wenn in dem definierenden Knoten Kontrollfluss modelliert ist. Listing 5.16 zeigt die ersten Schritte zur Transformation dieser Knoteninhalte. Zunächst werden alle Zustände aus dem definierenden Knoten in eine neue Region kopiert, siehe Zeile eins und zwei. Von jedem Zustand werden alle eingehenden (Zeile drei bis 13) und ausgehenden Transitionen (Zeile 14 bis 24) ebenfalls kopiert. Nachdem diese Inhalte kopiert wurden ist eine Überprüfung aller Transitionen notwendig. Es kann vorkommen, dass eine Transition entweder keinen Start- oder Zielzustand besitzt. In diesem Fall wird die Transition entfernt, da sie nicht weiter in ein Core-SCChart transformiert werden kann. Diese Überprüfung ist in Zeile 27 bis 39 zu sehen. Anschließend müssen noch die Eingangsvariablen durch die tatsächlich mit übergebenen Variablen ersetzt werden. Dies ist in Zeile 41 bis 49 des Listings zu sehen.

Nachdem die Eingangsvariablen ersetzt wurden, müssen wie üblich auch alle Ausgangsvariablen ersetzt werden. Wie in Unterkapitel 4.4 beschrieben, ist bei der Transformation von modelliertem Kontrollfluss in einem definierenden Knoten nicht nötig neue Zustände anzulegen. In Listing 5.17 ist der Programmausschnitt zur Ersetzung der Ausgangsvariablen dargestellt. Zunächst wird wie bei den bereits vorgestellten Varianten überprüft, ob die aktuelle Datenflussgleichung auf einen Knoten verweist und ob dieser Knoten dem aktuell transformierten aufrufenden Knoten entspricht. Falls der Ausgangswert schon einmal betrachtet wurde, dann müssen die entsprechenden Transitionen gesucht werden, in denen dieser Ausgangswert vorkam. In Zeile neun bis 15 des Listings ist die Suche implementiert. Dabei werden alle Zustände durchlaufen und alle dazugehörigen Transitionen. Wird dabei eine Transition gefunden, welche einen Effekt mit dem bereits betrachteten Ausgangswert enthält, so wird eine neue Zuweisung generiert. Diese neue Zuweisung, siehe Zeile 13 und 14, bekommt als Variable den aktuellen Ausgangswert und als Ausdruck eine Variablenreferenz auf die bereits vorgekommene Variable. Die neue Zuweisung wird der Transition als Effekt mit hinzugefügt. Wurde der aktuelle Ausgangswert noch nicht einmal betrachtet, so müssen trotzdem alle Transitionen aller Zuständen durchlaufen werden. Bei jeder Transition wird überpüft, ob ein Effekt die aktuelle Ausgangsvariable beinhaltet. Ist dies der Fall, so muss der Variablenwert ersetzt werden und die Transition mit dem Effekt wird gespeichert für eine später eventuell erneut vorkommende Verwendung dieses Ausgangswertes.

```
for (s: defNode.states) {
  val newState = s.copy
  for (t: s.incomingTransitions) {
    var newTrans = SCChartsFactory.eINSTANCE.createTransition
    if (transitionMapping.containsKey(t)) {
      newTrans = transitionMapping.get(t)
    } else {
      newTrans = t.copy
      transitionMapping.put(t, newTrans)
    }
    newTrans.targetState = newState
    newState.incomingTransitions += newTrans
  }
  for (t: s.outgoingTransitions) {
    var newTrans = SCChartsFactory.eINSTANCE.createTransition
    if (transitionMapping.containsKey(t)) {
      newTrans = transitionMapping.get(t)
    } else {
      newTrans = t.copy
      transitionMapping.put(t, newTrans)
    }
    newTrans.sourceState = newState
    newState.outgoingTransitions += newTrans
  }
  newRegion.states += newState
}
// remove transitions without source or target state
newRegion.states.forEach[ s|
  s.incomingTransitions.forEach[
    if (it.sourceState == null || it.targetState == null) {
      it.remove
    }
  ]
  s.outgoingTransitions.forEach[
    if (it.sourceState == null || it.targetState == null) {
      it.remove
    }
  ]
]
// replace input valued objects
defNode.inputs.forEach[ i|
  i.valuedObjects.forEach[ vo|
    newRegion.states.forEach[
      it.replaceAllOccurrences(vo,
        (callNode.parameters.get(i.valuedObjects.indexOf(vo))
                    as ValuedObjectReference).valuedObject)
    ]
  ]
]
```

Listing 5.16. Codeausschnitt zur Transformation von aufrufenden Knoten, Teil 4

```
for (eq: dataflow.equations) {
  if (eq.node != null) {
    if (eq.node.equals(callNode)) {
      val key = callNode.id + "."
              + (eq.expression as ValuedObjectReference).valuedObject.name
      val vo = eq.valuedObject
      val voRef = (eq.expression as ValuedObjectReference).valuedObject

      if (voMapping.containsKey(key)) {
        for (s: newRegion.states) {
          for (t: s.allContainedTransitions) {
            if (transitionsMapping.containsKey(key+t.id)) {
              val newAssignment = createNewAssignment(vo,
                                  voMapping.get(key).reference)
              transitionsMapping.get(key+t.id).effects += newAssignment
            }
          }
        }
      } else {
        for (s: newRegion.states) {
          for (t: s.allContainedTransitions) {
            for (assign: t.allContainedAssignments) {
              if (assign.valuedObject.equals(voRef)) {
                assign.valuedObject = vo
                voMapping.put(key, vo)
                transitionsMapping.put(key + t.id, t)
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 5.17. Codeausschnitt zur Transformation von aufrufenden Knoten, Teil 5

5.3.3. Transformation von Datenflussgleichungen

Nach der Transformation aller Knoten werden als Letztes noch alle Datenflussgleichungen transformiert. Da bereits alle Gleichungen, die auf einen Ausgangswert eines Knotens verweisen, während der Knotentransformationen abgearbeitet wurden, bleiben nur noch Gleichungen der bekannten Form $x = e$ übrig. Der dazugehörige Ausschnitt der Referenztransformation ist in Listing 5.18 dargestellt. Als vorbereitender Schritt wird für alle Datenflussgleichungen eine neue Zuweisung generiert und diese in einer Liste, assignmentList, gespeichert. Dieser Vorbereitungsschritt ist in Zeile eins bis acht des Listings zu sehen. Wenn diese Liste nicht leer ist, so wird eine neue Region im Elternzustand der

```
1  val assignmentList = <Assignment> newArrayList
2  for (eq: dataflow.equations) {
3    if (eq.node == null) {
4      // Equation: eq.node == null
5      // and create new assignment as transition effect
6      assignmentList += createNewAssignment(eq.valuedObject, eq.expression)
7    }
8  }
9
10 if (!assignmentList.empty) {
11   // => create new region with initial and final state for each expression
12   val rRegion = parentState.createRegion("_" + dataflow.id + regionCounter)
13   rRegion.label = dataflow.label + regionCounter
14
15   val newState = rRegion.createState("_" + dataflow.ID + idCounter)
16   newState.label = dataflow.label + idCounter + "_start"
17   newState.setInitial
18
19   val newState2 = rRegion.createState("_" + dataflow.id + idCounter)
20   newState2.label = dataflow.label + idCounter + "_end"
21   newState2.setFinal
22
23   regionCounter = regionCounter + 1
24   idCounter = idCounter + 1
25
26   val transition = createNewTransition(newState, newState2)
27   transition.setImmediate
28
29   for (assign: assignmentList) {
30     transition.effects += assign
31   }
32 }
```

Listing 5.18. Codeausschnitt zur Transformation von Datenflussgleichungen

Datenflussumgebung generiert. Innerhalb dieser Region werden erneut ein Start- und ein Endzustand erzeugt. Eine instantane Transition zwischen diesen beiden Zuständen wird generiert und als Effekte werden alle neuen Zuweisungen dieser Transition hinzugefügt. Dies ist in Zeile zehn bis 32 des Listings dargestellt.

Nachdem alle Datenflusselemente transformiert wurden, ruft sich die Referenztransformation rekursiv auf. Dies ist notwendig, damit eventuell weiter verschachtelte, durch Referenzen hinzugekommene, Datenflussumgebungen ebenfalls mit transformiert werden. Abschließend werden in einem letzten Schritt noch alle leeren Umgebungen gelöscht, das heißt, wenn keine Zustände mehr in diesen Umgebungen enthalten sind.

Kapitel 6

Auswertung

Dieses Kapitel stellt die Ergebnisse meiner Implementierung von Datenfluss in SCCharts vor. Dazu werden unterschiedliche Beispielmodelle sowohl textuell als auch graphisch angeführt. Das Unterkapitel 6.1 zieht dabei einen Vergleich zu den Beispielmodellen aus der Einleitung meiner Arbeit. In Unterkapitel 6.2 werden weitere Modellierungsmöglichkeiten ausgewertet. Abschließend wird in Unterkapitel 6.3 eine Auswertung in Bezug auf einen Anwendungsfall genommen, das Modelleisenbahnpraktikum[1] von 2007 am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme.

6.1. Vergleich der Ergebnisse

In diesem Unterkapitel werden Beispielmodelle gezeigt und bewertet, die den Beispielen der Ausgangspunkte meiner Arbeit nachempfunden sind. Beginnend liefert Abschnitt 6.1.1 einen Vergleich zwischen der klassischen SCChart Notation und der in dieser Arbeit vorgestellten Notation zur Beschreibung von Datenfluss in SCCharts.

6.1.1. Notationsvergleich

Als erstes betrachte ich noch einmal das Beispiel aus Kapitel 4, in dem ein separat als SCChart modelliertes Flipflop im Datenfluss referenziert wird. Die graphische Repräsentation ist annähernd gleich geblieben, der benötigte textuelle Modellierungsumfang ist aber deutlich kompakter. Wie auf der rechten Seite der Abbildung 6.1 zu sehen ist, werden nur elf statt 16 Zeilen für das Modell benötigt. Die Einsparung liegt dabei in der kürzeren Notation für Datenflusselemente, wie ab Zeile acht zu sehen ist. Diese Einsparung in der textuellen Beschreibung macht bei mittleren und großen Modellen einen signifikanten Unterschied im Modellierungsumfang aus.

In Abbildung 6.2 ist die Visualisierung des Modells zu sehen. Die obere Umgebung beinhaltet exemplarisch Kontrollfluss, die untere Umgebung beinhaltet Datenfluss, hier in Form eines referenzierenden Knotens.

[1] http://www.informatik.uni-kiel.de/rtsys/modelleisenbahn/

```
scchart minimal {
  input bool A, B, R;
  output bool O = false;
  region:
    initial state A
    --> B;
    final state B;
  dataflow:
    node A "A" input: A
    --> RegA.S;
    node R "R" input: R
    --> RegA.R;
    node RegA "RegA": flipflop
    O --> O;
    node O "O" output: O;
}
```

(a) Beispielmodell mit der klassischen SCChart-Notation für Datenfluss

```
scchart minimal {
  input bool A, B, R;
  output bool O = false;
  region:
    initial state A
    --> B;
    final state B;
  dataflow:
    RegA "RegA" = ref flipflop(A, R);
    O = RegA.O;
}
```

(b) Beispielmodell mit der in dieser Arbeit vorgestellten Datenflussnotation

Abbildung 6.1. Vergleich des Modellierungsumfangs anhand eines Referenzknotens

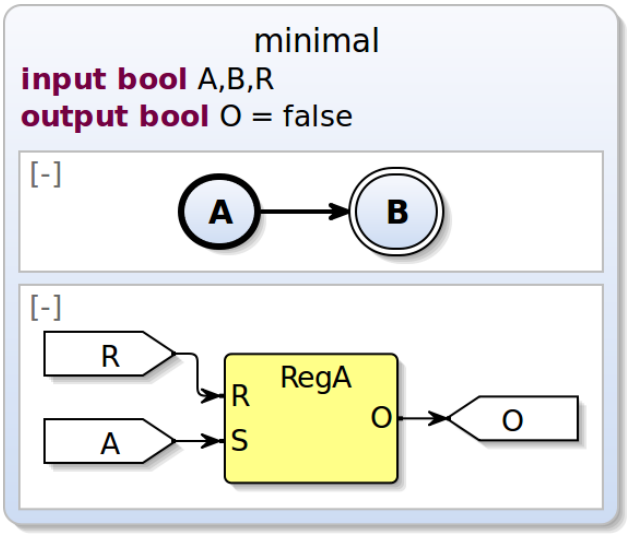


Abbildung 6.2. Ein SCChart mit Daten- und Kontrollfluss

6.1.2. Einleitungsbeispiele

In diesem Abschnitt greife ich die beiden einleitenden Beispiele aus Kapitel 1 nochmals auf. Die dort vorgestellten Abbildungen 1.2a für Kontrollfluss und 1.2b für Datenfluss werden hier in einem einzelnen SCChart zusammengefasst. Abbildung 6.3 zeigt die Visualisierung dieses SCCharts. Beide Modellierungskonzepte

können in einem Modell parallel beschrieben und verwendet werden. Die obere Umgebung enthält den Kontrollfluss. Die untere Umgebung enthält die beiden Datenflussgleichungen x = 2 * y + z und w = x + 1.

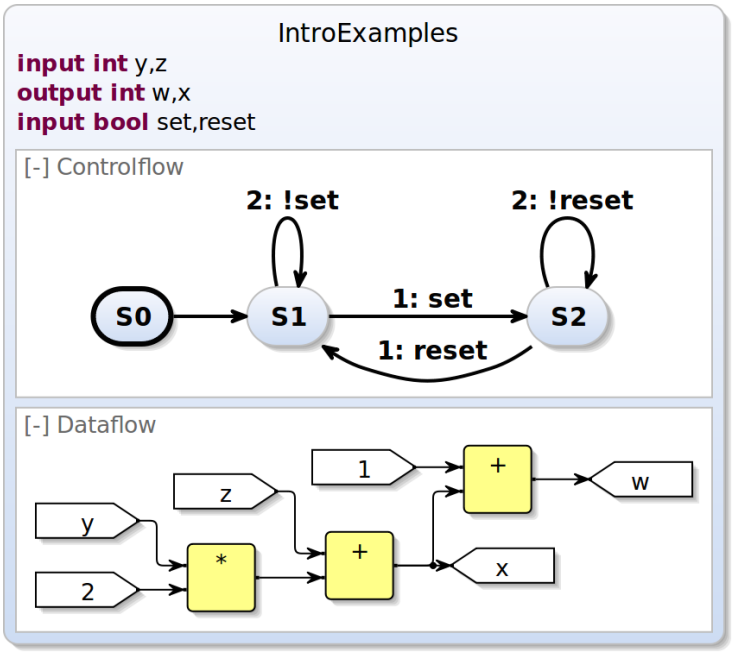


Abbildung 6.3. Ein SCChart mit einer Daten- und einer Kontrollflussumgebung, die den beiden separaten Diagrammen aus dem Einleitungskapitel nachempfunden sind.

6.1.3. Vergleich zu einem SCADE Modell

Ein Hauptziel meiner Arbeit, die kombinierte Verwendung von Daten- und Kontrollfluss in SCCharts, lässt sich wie im letzten Beispiel bereits dargestellt modellieren. In diesem Abschnitt gehe ich etwas näher auf einen Vergleich zur Modellierungssprache SCADE ein. In Abbildung 1.3 wurde ein Ausschnitt eines SCADE-Modells vorgestellt, das beide Modellierungskonzepte vereint. In Abbildung 6.4 ist ein SCChart zu sehen, das diesem vorgestellten Modellausschnitt nachempfuden ist. Zu sehen sind zwei Zustände (also Kontrollfluss), in denen jeweils Datenfluss modelliert ist. Die Berechnung des Ausgangswertes tCmd ist dabei abhängig von dem jeweiligen Zustand, in dem sich das System befindet. Die jeweiligen Datenflusskomponenten Regulator im Zustand RegulOn und MngThrottle im Zustand StandBy stellen diese Berechnung als Aufruf eines definierenden Knotens dar. Das Verhalten innerhalb dieser Knoten kann nun ebenfalls entsprechend dem Verhalten der Datenflusskomponenten aus dem SCADE-Modell nachmodelliert werden.

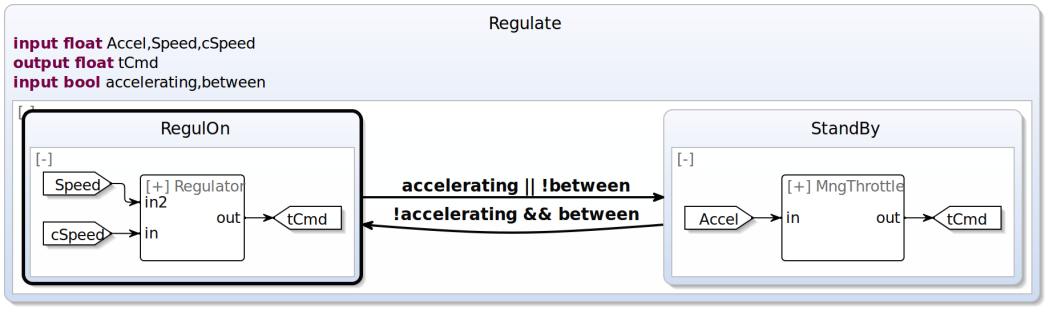


Abbildung 6.4. Beispiel zur kombinierten Modellierung von Daten- und Kontrollfluss in einem SCChart.

6.2. Weiterere Ergebnisse der Implementierung

Dieses Unterkapitel zeigt weitere Modellierungsmöglichkeiten mit meiner Implementierung und bewertet die erreichte Funktionalität.

6.2.1. Funktionsumfang

Das vorherige Unterkapitel hat gezeigt, dass die Modellierung von Datenflussgleichungen auf oberster Ebene und innerhalb von Zuständen implementiert wurde. Im Folgenden wird die Verwendung der verschiedenen Knotentypen weiter untersucht und ausgewertet.

Das nächste Beispiel zeigt ein SCChart mit dem Aufruf eines definierenden Knotens, sowie der Weiterverarbeitung eines der Ausgangswerte dieses Knotens. In Listing 6.1 ist das textuelle Modell dargestellt. Zeile fünf bis acht zeigt die Definierung eines Knotens, der zwei Datenflussgleichungen enthält. Die beiden

```
1  scchart ExampleDataflow {
2    input int in1, in2;
3    output int out1, out2, out3;
4    dataflow df:
5      node compute(int a, b) returns (int x, y) {
6        x = (2 * a) + b;
7        y = a - b;
8      }
9    call = compute(in1, in2);
10   out1 = call.x;
11   out2 = call.y;
12   out3 = 10 * out2;
13 }
```

Listing 6.1. Textuelles Modell einer Datenflussumgebung mit Knoten und Datenflussgleichung

Ausgangswerte out1 und out2 greifen auf jeweils einen Ausgangswert des Knotens zu. Der Ausgangswert out3 berechnet sich aus der Gleichung 10 * out2. Das dazugehörige visualisierte Modell ist in Abbildung 6.5 dargestellt. Der Aufruf des Knotens compute ist hier expandiert zu sehen, damit die darin modellierten Datenflussgleichungen ebenfalls zu erkennen sind. Die Berechnung des Ausgangswertes out3 hängt wie deutlich zu sehen von einem Ausgangswert des Knotens ab. Dies wird korrekt visualisiert, in dem der benötigte Eingang für die Kante nicht direkt auf out2 gelegt wird, sondern auf den Ausgangswert y des Knotens.

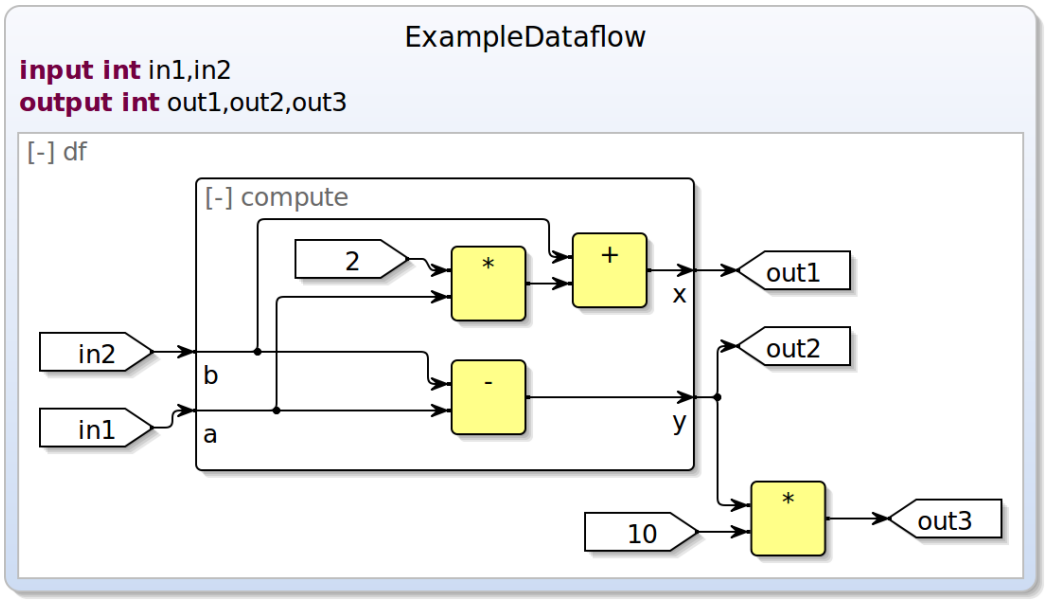
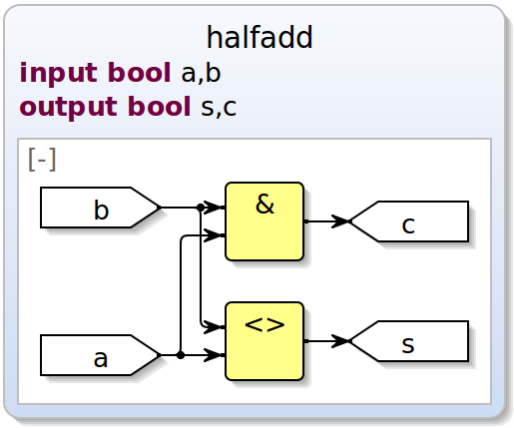


Abbildung 6.5. Datenflussumgebung mit einem definierenden Knoten und dessen Aufruf (expandierte Darstellung), sowie Weiterverarbeitung eines Ausgangswertes des Knotens.

Das nächste Beispiel befasst sich mit der Beschreibung eines Volladdieres, der aus zwei Halbaddierern aufgebaut ist. Der Halbaddierer ist als ein SCChart mit dem Namen halfadd modelliert. In Abbildung 6.6 ist die textuelle Beschreibung auf der rechten Seite und die entsprechende Visualisierung auf der linken Seite zu sehen. Die beiden booleschen Eingangswerte a und b berechnen das Summenbit s und den Übertrag c. Die XOR-Operation zur Berechnung des Summenbits wird hier durch den Ungleich-Operator (<>) dargestellt.

Ein zweites SCChart mit dem Namen add ist in Listing 6.2 beschrieben. In diesem SCChart werden die beiden benötigten Halbaddierer über die referenzierenden Knoten ha1 und ha2 eingebunden. Da bei dem Aufruf eines referenzierenden Knotens keine Ausdrücke mit übergeben werden können, ist eine temporäre Variable tmp_s als Zwischenergebnis für das Summenbit des ersten Halbaddierers notwendig. Ebenso verhält es sich bei der Berechnung des Übertrags co, so dass zwei weitere temporäre Variablen c1 und c2 verwendet werden. Diese



(a) Graphisches Modell des Halbaddierers

```
1 scchart halfadd {
2   input bool a, b;
3   output bool s, c;
4
5   dataflow:
6     s = a <> b;
7     c = a & b;
8 }
```

(b) Textuelles Modell des Halbaddierers

Abbildung 6.6. Ein Halbaddierer als SCChart

nur als Zwischenschritt benötigten Variablen werden als lokale Variablen in der Datenflussumgebung in Zeile sechs des Listings deklariert.

```
1  scchart add {
2    input bool a, b, ci;
3    output bool s, co;
4
5    dataflow:
6      output bool tmp_s, c1, c2;
7
8      ha1 = ref halfadd(a, b);
9      tmp_s = ha1.s;
10     ha2 = ref halfadd(tmp_s, ci);
11     c1 = ha1.c;
12     c2 = ha2.c;
13     s = ha2.s;
14     co = c1 | c2;
15 }
```

Listing 6.2. Textuelle Beschreibung eines Volladdierers

Die graphische Repräsentation des Addierers als SCChart ist in Abbildung 6.7 dargestellt. Wie zu erkennen ist, kann ein separates SCChart mehrfach aus einem anderen SCChart heraus referenziert werden. Diese Modularität trägt erheblich zur besseren Lesbarkeit und Wartbarkeit von großen Modellen bei. Der Volladdierer lässt sich korrekt modellieren und beschreiben. Durch die mit meiner Implementierung benötigten Zwischenschritte wirkt das Modell leider noch nicht gänzlich minimiert. Die Visualisierung der für die Zwischenschritte notwendigen Variablen schränkt die Übersichtlichkeit des Modells ein. An dieser Stelle wäre es

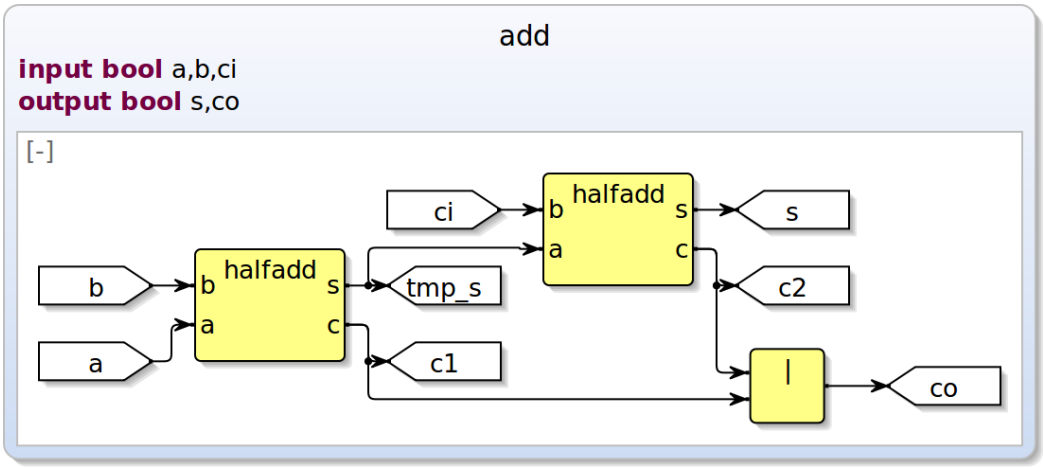


Abbildung 6.7. Ein Volladdierer als SCChart

sinnvoll die Implementierung dahingehend zu erweitern, dass auch Ausdrücke als Parameter bei einem Knotenaufruf mit übergeben werden können. Dadurch würde man zum Beispiel Zeile neun des Listings 6.2 einsparen und könnte in Zeile zehn direkt den Summenausgang des ersten Halbaddierers mit übergeben. Die Referenzierung des zweiten Halbaddierers ließe sich dann wie folgt beschreiben: `ha2 = ref halfadd(ha1.s, ci);`. In der Visualisierung wäre dann kein Ausgangssymbol für diesen Zwischenwert mehr zu sehen.

In einem letzten Beispiel für diesen Abschnitt der Auswertung zeige ich ein etwas umfangreicheres Modell, welches verschiedene der vorgestellten Funktionen beinhaltet. In Abbildung 6.8 ist ein SCChart mit drei Zuständen zu sehen. Ausgehend vom Startzustand `init` gelangt man entweder in den Zustand `state1` oder `state2`. Der erste Zustand enthält Datenfluss, einmal eine Datenflussgleichung und zwei Aufrufe des definierenden Knotens `test`. Dabei ist zu beachten, dass für den zweiten Aufruf des Knotens die Reihenfolge der beiden übergebenen Parameter `I` und `I2` vertauscht wurde. Die Datenflussgleichung enthält die Negation eines Eingangswertes, verodert mit dem Ausgangswert eines Knotens. Der zweite Zustand enthält wiederum drei Zustände. Der Startzustand `s1` enthält zwei nebenläufige Umgebungen mit Datenfluss. Die obere Umgebung in diesem Startzustand enthält eine Datenflussgleichung, die untere Umgebung den Auruf eines definierenden Knotens, welcher Kontrollfluss enthält. Dieses Beispiel verdeutlicht noch einmal die beliebige Kombination und Verschachtelung von Daten- und Kontrollfluss.

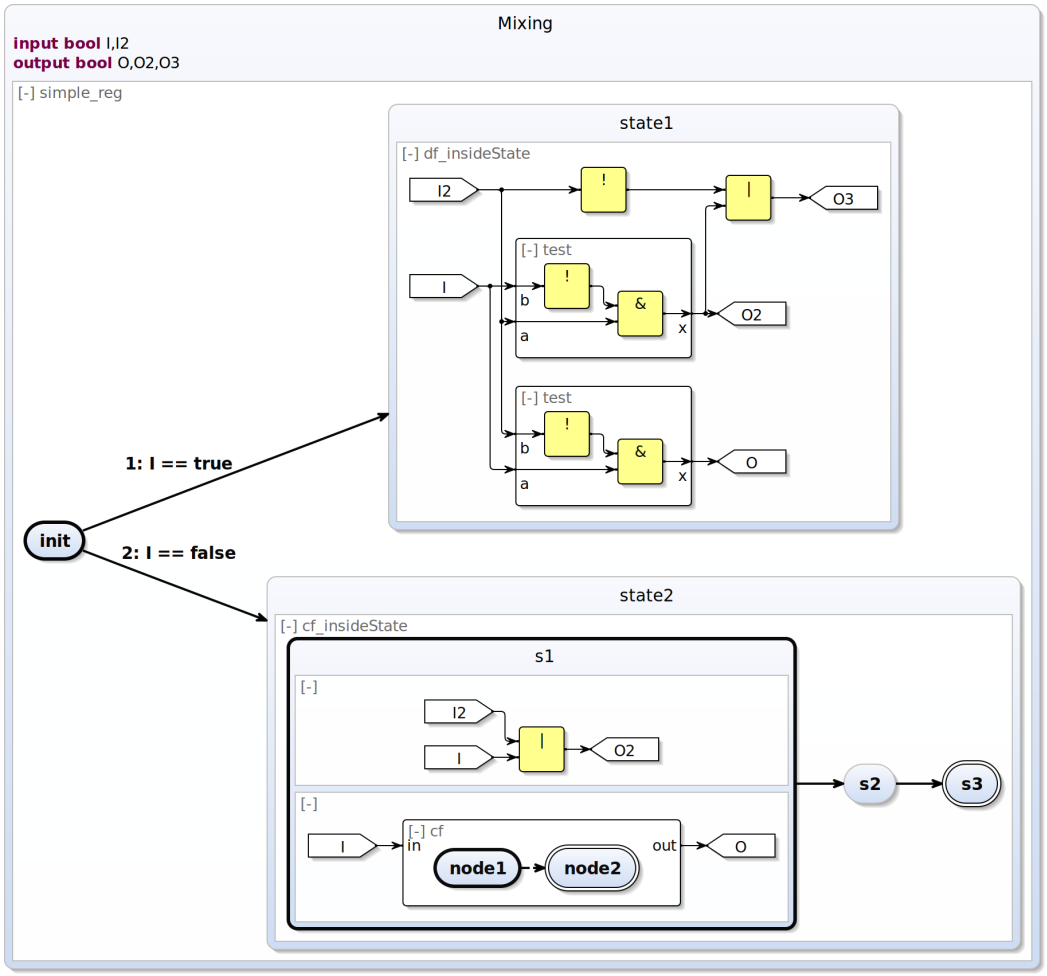


Abbildung 6.8. Beispiel eines SCCharts mit verschachteltem Daten- und Kontrollfluss

6.2.2. Transformationsbeispiele

Nach der Auswertung der Funktionalität der Datenflussmodellierung in SCCharts folgt in diesem Abschnitt die Auswertung der Referenztransformation. Dazu betrachte ich als erstes erneut das SCChart aus Abbildung 6.3. In diesem SCChart sind die beiden Einleitungsbeispiele der Modellierungskonzepte nebenläufig modelliert. Nach der Ausführung der Referenztransformation gibt es zwei parallele Regionen, welche beide Kontrollfluss enthalten. In Abbildung 6.9 ist dieses transformierte SCChart zu sehen. Die untere Region ist die unveränderte Kontrollflussregion des Ausgangsmodells. Die obere Region enthält den transformierten Datenfluss.

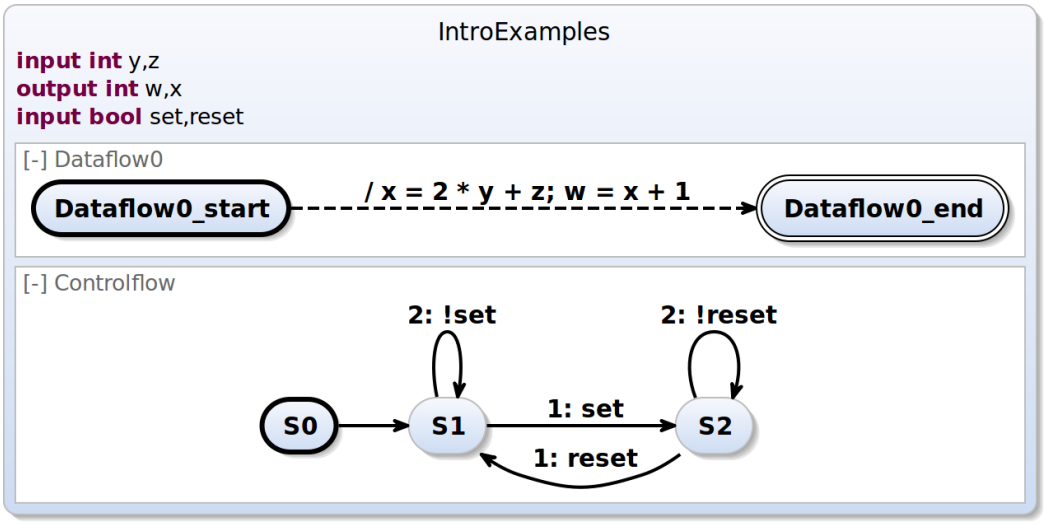


Abbildung 6.9. Das transformierte SCChart aus Abbildung 6.3

Als nächstes Beispiel zeige ich die Transformation des SCCharts aus Listing 6.1, welches in Abbildung 6.5 dargestellt ist. In diesem Modell werden ein Knoten und eine Datenflussgleichung transformiert. Das transformierte SCChart ist in Abbildung 6.10 zu sehen. Wie in Abschnitt 4.4.2 beschrieben, wird für jeden Knoten bei der Transformation eine neue separate Region generiert. Wie zu erkennen ist, funktioniert die Transformation des Datenflusses wie beschrieben, die Effekte der instantanen Transitionen entsprechen den Datenflussgleichungen. Die Generierung einer eigenen nebenläufigen Region (`df1`) für den transformierten Knoten ist an dieser Stelle aber nicht zwingend notwendig. Der Effekt in dieser Region könnte stattdessen der Effektliste der Transition aus Region `df0` mit hinzugefügt werden.

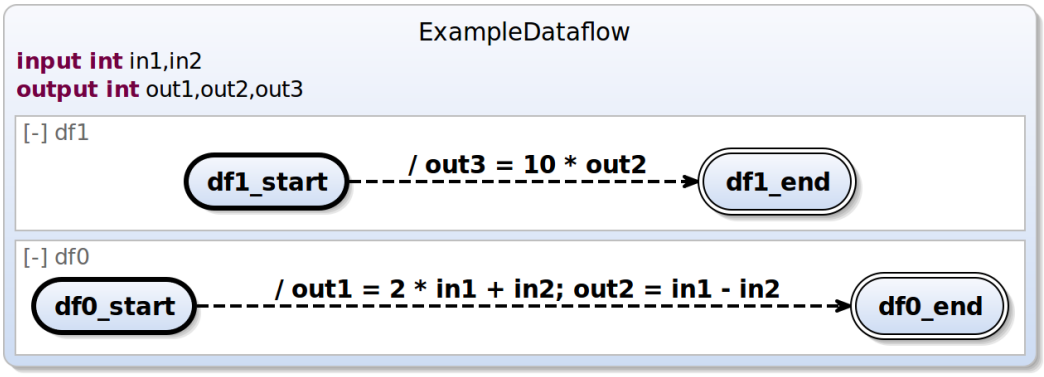


Abbildung 6.10. Das transformierte SCChart aus Abbildung 6.5

6.3. Anwendungsfall Modelleisenbahnprojekt

Ein weiterer Punkt in meiner Auswertung ist es aufzuzeigen, inwieweit die Modelle des Modelleisenbahnpraktikums von 2007 mit der vorgestellten Implementierung von Datenfluss in SCCharts bereits umzusetzen sind. Im Praktikum wurden die Modelle mit SCADE beschrieben. Der Hauptaspekt zur Modellierung des Modelleisenbahnpraktikums in SCCharts, die kombinierte Beschreibung von Daten- und Kontrollfluss, ist mit dieser Implementierung, wie in den vorangegangenen Unterkapiteln beschrieben, möglich.

Um die Modelle aus diesem Praktikum in SCCharts gänzlich beschreiben zu können fehlt es noch an Folgendem. Ein wichtiger und nützlicher Aspekt wäre die Möglichkeit der Fallunterscheidung in Form einer if-then-else Grammatikregel. Zu Übersichtszwecken wäre ein switch Operator ebenfalls sinnvoll. Der switch Operator könnte alternativ auch durch mehrere if-then-else Anweisungen dargestellt werden. In SCADE lassen sich eigene Datenstrukturen definieren. Des Weiteren gibt es diverse Operatoren, mit denen man unter anderem einzelne Elemente dieser Strukturen auslesen und beschreiben kann. Für den Datenfluss in SCCharts ließe sich solch ein Konzept von Datenstrukturen bisher nur auf Arrays abbilden, wenn alle Elemente der Datenstruktur vom gleichen Typ sind.

SCADE Modelle sind modular aufgebaut, das heißt ein großes Modell kann durch die Beschreibung eigener Operatoren in Teilmodelle zerlegt werden. Wie im vorherigen Unterkapitel aufgezeigt, bietet die Definierung von Knoten und Referenzierung von separaten SCCharts eine benutzerfreundliche modulare Entwicklung. Als ein Beispiel zur Verdeutlichung ist in Abbildung 6.11 der SCADE Operator AllTrainsInStationTest aus dem Praktikum abgebildet. Ein nachempfun-

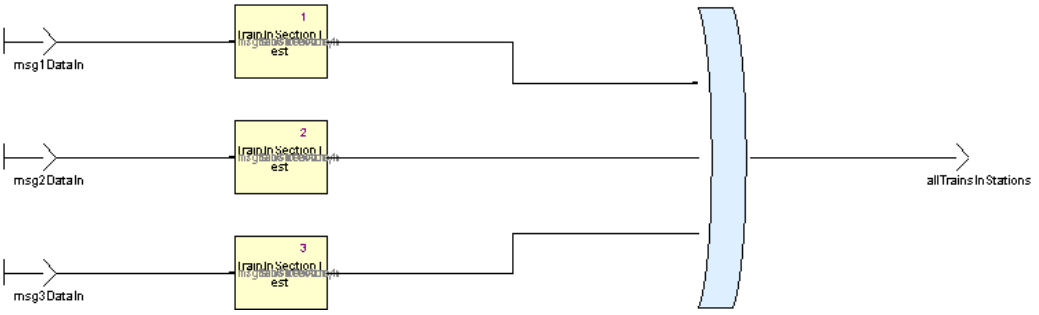
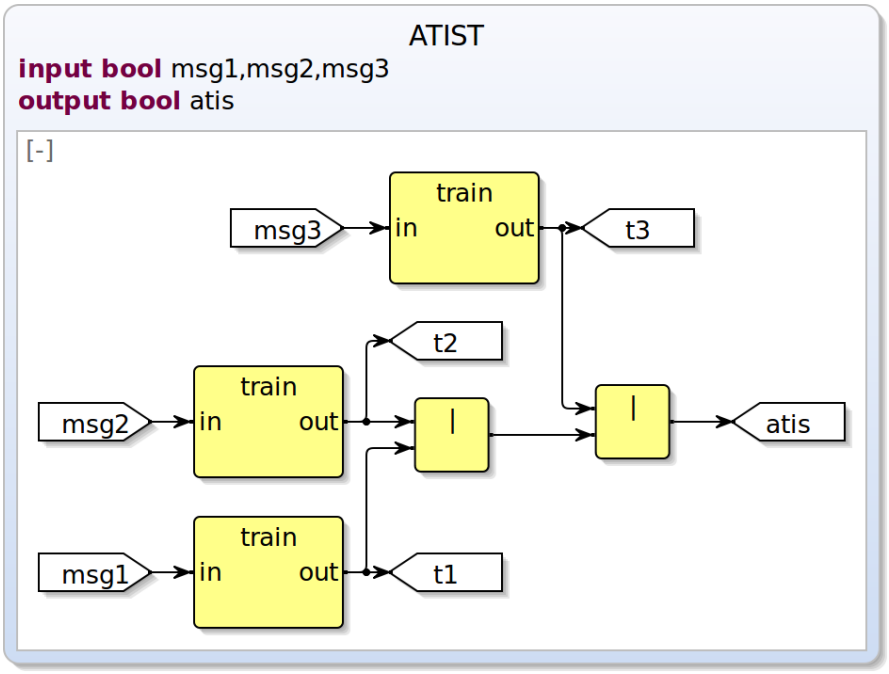
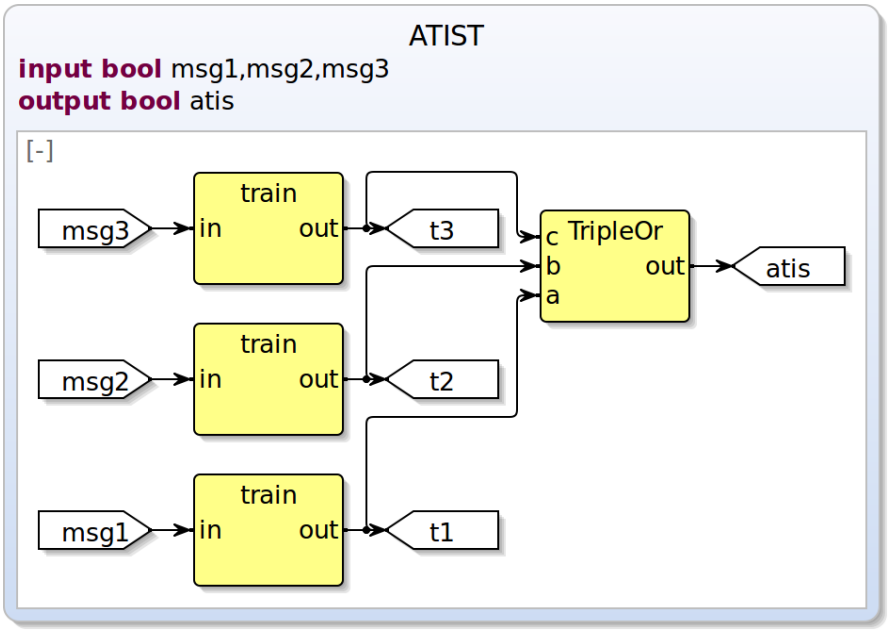


Abbildung 6.11. SCADE Operatormodell: AllTrainsInStationTest (Praktikumsreport)

denes SCChart ist in Abbildung 6.12a zu sehen. Der in dem Modell verwendete Operator trainsInSectionTest ist hier mit dem Namen train abgekürzt und als ein separates SCChart modelliert. Wie im Vergleich zu erkennen ist, gibt es in SCADE die Möglichkeit boolesche Operatoren mit mehr als zwei Eingängen zu verwenden. Die in SCCharts verwendeten KExpressions zur Beschreibung von Ausdrücken



(a) SCChart mit hintereinander geschalteten Oder Operatoren



(b) SCChart mit dreifach Oder als Referenzknoten

Abbildung 6.12. Nachempfundenes SCChart zum Operator AllTrainsInStationTest

erlauben ebenfalls die Verwendung von booleschen Operatoren mit mehr als zwei Eingängen. Die interne Struktur baut dies jedoch aus hintereinander geschalteten Operatoren mit jeweils zwei Eingängen auf, so dass in dem SCChart zwei Elemente für die Veroderung dargestellt sind. Es gibt mehrere Möglichkeiten, die Darstellung auf ein einzelnes Element zu reduzieren. Eine Variante wäre es, die Grammatik der KExpressions entsprechend anzupassen. Eine andere Möglichkeit besteht in der Modellierung eines SCCharts, welches die Veroderung von drei statt zwei Eingängen beinhaltet. Dieses separate SCChart kann dann referenziert werden, wie es in Abbildung 6.12b zu sehen ist.

In einem weiteren Beispiel ist in Abbildung 6.13 ein Ausschnitt des SCADE Operators ScheduleEditor des Praktikums zu sehen. Dieses Beispiel veranschaulicht die kombinierte Modellierung von Daten- und Kontrollfluss, sowie die bereits angesprochene Modularität von Modellen. Zu sehen ist der Kontrollfluss innerhalb der SSM SM2, bestehend aus zwei Zuständen. Des Weiteren gibt es den Datenfluss, der den Wert choosenTrain und das Signal pushedInfinite berechnet. In Abbildung 6.14 ist ein SCChart dargestellt, welches diesem Operatorausschnitt nachempfunden ist. Die untere Umgebung in dem SCChart enthält den Kontrollfluss aus der SSM

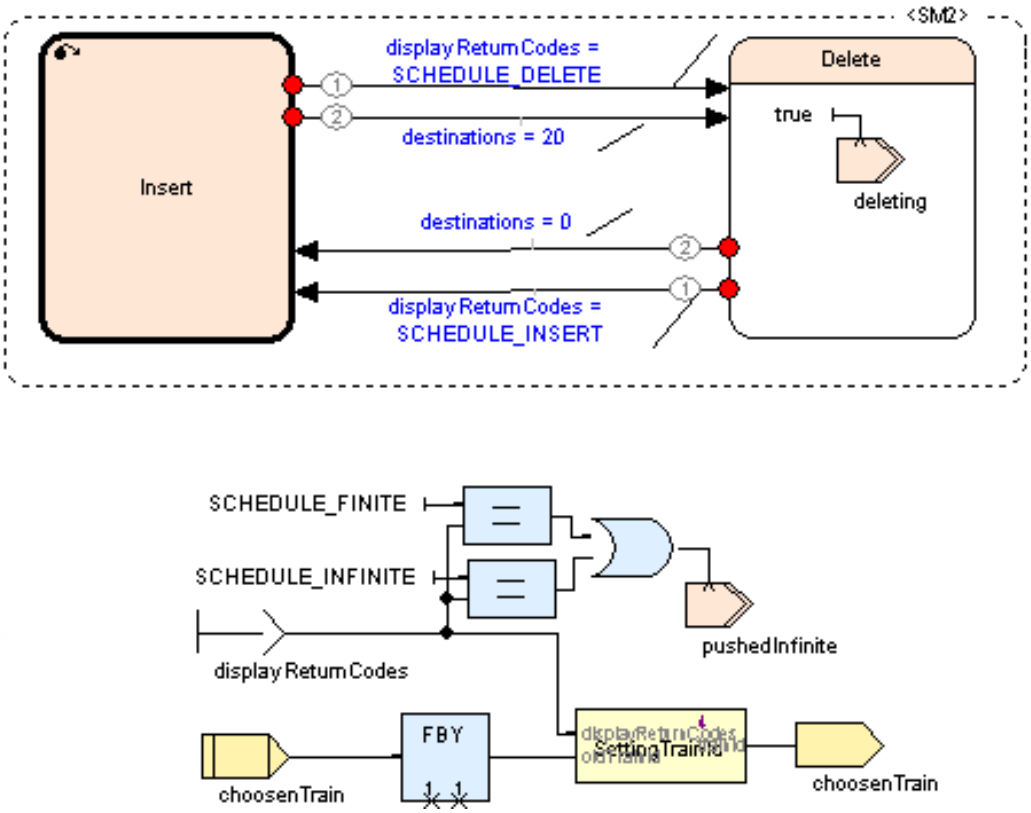


Abbildung 6.13. Ausschnitt SCADE Operatormodell: ScheduleEditor (Praktikumsreport)

des SCADE Operators. Die obere Umgebung enthält den Datenfluss. Der SCADE Operator FBY (followed by) und der selbst definierte Operator SettingTrainNo sind in dem SCChart als Aufruf eines definierenden Knotens dargestellst. Alternativ könnte man auch zwei separate SCCharts referenzieren. Um das SCChart mit der Textlänge von Labeln nicht zu überladen, wurden einige Label in einer kürzeren Form verwendet.

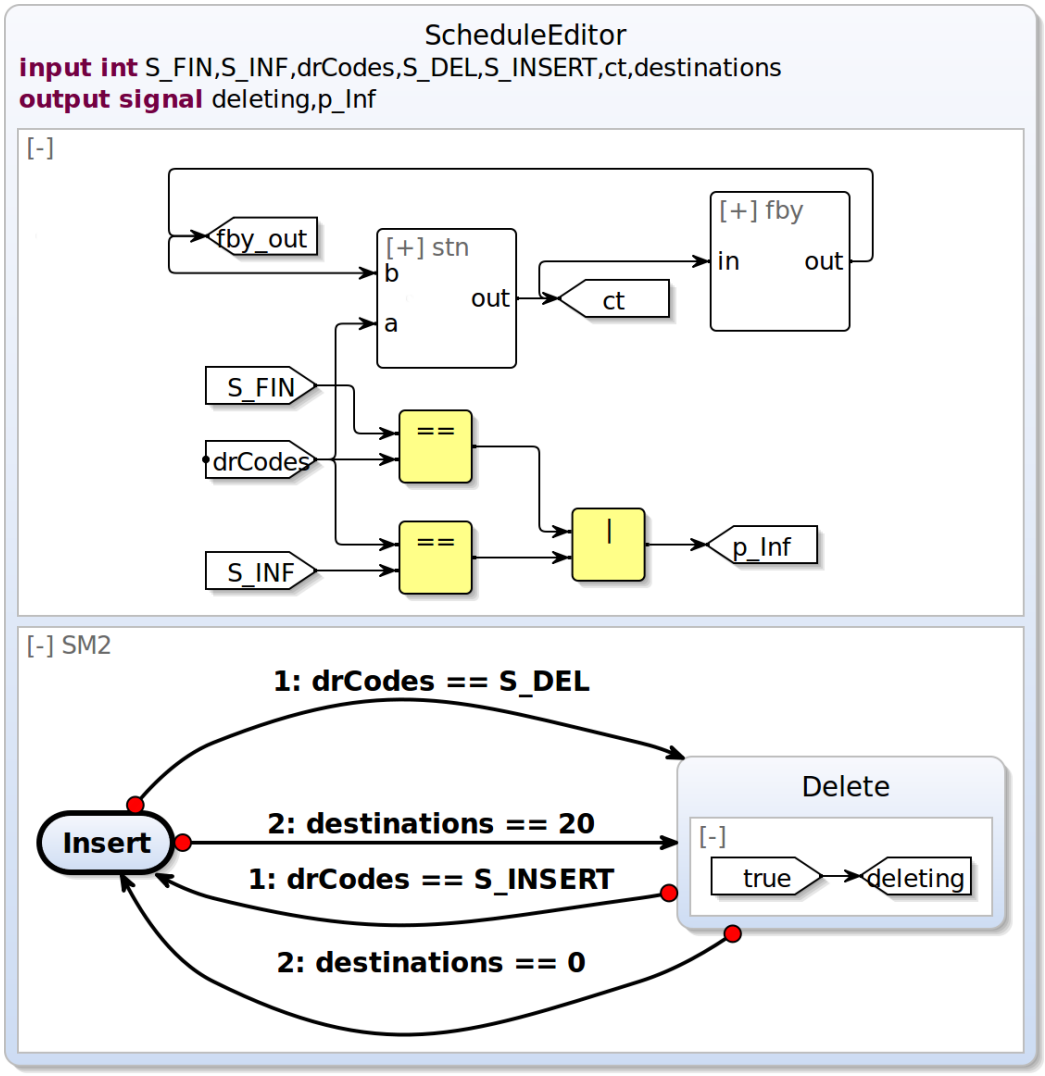


Abbildung 6.14. Nachempfundenes SCChart zum ScheduleEditor (Ausschnitt)

Kapitel 7

Schlussbetrachtung

Als Abschluss meiner Diplomarbeit werden die erreichten Ziele in Unterkapitel 7.1 zusammengefasst. Danach wird in Unterkapitel 7.2 noch ein Ausblick über mögliche weiterführende Arbeiten gegeben.

7.1. Zusammenfassung

Das Ziel meiner Diplomarbeit war die Erweiterung von SCCharts um Datenfluss. Dabei sollte die bisherige Funktionalität der SCCharts nicht beeinträchtigt werden. Durch das Hinzufügen einer neuen nebenläufigen Umgebung zur Modellierung von Datenfluss ist dies gelungen.

Mit meinem Konzept ist es möglich komplexe Datenflussgleichungen zu modellieren, anzeigen zu lassen und mittels einer Modelltransformation in klassische SCCharts zu übersetzen. Neben der Modellierung von Datenflussgleichungen der Form $x = e$, mit x als Variable und e als Ausdruck zur Berechnung des Variablenwertes, ist auch die Verwendung von Datenflussknoten möglich. Das bedeutet, dass in einem Knoten ein bestimmtes Verhalten, sei es als Daten- oder Kontrollfluss ausgedrückt, modelliert werden kann. Dieses einmalig beschriebene Verhalten kann dann in der Datenflussumgebung beliebig oft aufgerufen und verwendet werden. Dadurch ist eine einfachere und übersichtlichere Modellierung und Wartung der Modelle gegeben. Des Weiteren ist es möglich, Daten- und Kontrollfluss beliebig hierarchisch zu verschachteln. Es kann in einem Zustand Daten- und Kontrollfluss modelliert werden, welcher wiederum Daten- und Kontrollfluss in separaten Umgebungen enthalten kann.

Die Visualisierung von Datenfluss fügt sich nahtlos in die graphische Repräsentation von SCCharts ein. Die Umgebungen zur Modellierung von Daten- und Kontrollfluss können beliebig parallel verwendet werden. Dabei wird der Kontrollfluss wie gewohnt als Zustandsdiagramm dargestellt und der neu hinzugefügte Datenfluss im Stil von Blockdiagrammen.

Die grundlegende Funktionalität der Transformation von Datenfluss in klassische SCCharts wurde ebenfalls implementiert. Alle Datenflussumgebungen werden während der Referenztransformation in klassische Regionen der SCCharts transformiert. Die modellierten Datenflussgleichungen werden übersetzt in Effekte von instantanen Transitionen zwischen Zuständen in den transformierten Regionen.

7.2. Ausblick

Dieses letzte Unterkapitel beschäftigt sich mit einigen Vorschlägen für weiterführende Arbeiten, die die Modellierung von Datenfluss in SCCharts vereinfachen und um neue Möglichkeiten erweitern. Des Weiteren gibt es einen Ausblick zur Erweiterung der Referenztransformation, welche aufgrund ihrer Komplexität in dieser Arbeit nur in einer grundlegenden Form implementiert wurde.

7.2.1. Erweiterungen und Vereinfachungen der Datenflussmodellierung

Dieser Abschnitt beschreibt einige Möglichkeiten zur Erweiterung und Vereinfachung der Datenflussmodellierung in SCCharts und liefert erste Konzeptideen zu deren Implementierung.

Fallunterscheidung in einer Datenflussgleichung Bisher ist es noch nicht möglich Datenflussgleichungen mit einer Fallunterscheidung zu verwenden. Mein Vorschlag zur Modellierung ist in Anlehnung an Lustre:

$$x = \text{if} < c > \text{then} < e_1 > \text{else} < e_2 >.$$

Dabei ist x eine Variable, c ein Ausdruck der einen booleschen Wert liefert und e_1 und e_2 sind zwei Ausdrücke, die den Wert von x in Abhängigkeit von c bestimmen. Für die Visualisierung kann dabei ein neues Knotenelement eingeführt werden, welche neben den beiden Ausdrücken als Eingangskanten auf der linken Seite noch eine weitere Eingangskante auf der oberen Seite des Knotens bekommt, die den Ausdruck der Bedingung c darstellt. Zur Verdeutlichung ist in Abbildung 7.1 eine Möglichkeit zur Darstellung einer Fallunterscheidung zu sehen. Die Referenztransformation müsste dann entsprechend der Fallunterscheidung noch angepasst werden. Die Effekte einer Transition die eine Zuweisung mit Fallunterscheidung enthalten könnten wie folgt beschrieben werden: $/x = c?\ e_1;\ e_2$. Wie bereits beschrieben wird der Variablen x entsprechend der Bedingung c dann der Ausdruck e_1 oder e_2 zugewiesen.

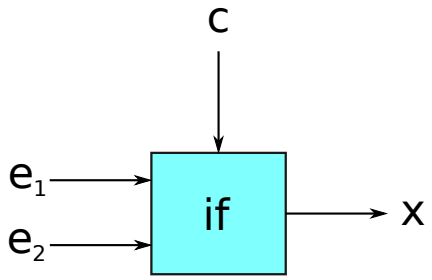


Abbildung 7.1. Beispiel zur graphischen Darstellung einer Fallunterscheidung

Aufrufende Knoten als Zwischenschritt aufheben Die in dieser Arbeit vorgestellte Implementierung von Datenfluss in SCCharts erwartet für die Verwendung von definierenden Knoten den Zwischenschritt eines aufrufenden Knotens. Eine mögliche Vereinfachung an dieser Stelle ist es, den Zwischenschritt wegzulassen. Eine Datenflussgleichung könnte dann direkt auf die Ausgänge eines definierenden Knotens zugreifen. In Abbildung 7.2 ist beispielhaft die Definierung eines Knotens compute zu sehen. Die aktuelle Notation ist auf der linken Seite dargestellt, der Zwischenschritt über einen aufrufenden Knoten ist in Zeile neun ersichtlich. Auf der rechten Seite sieht man eine mögliche Notation zum Zugriff auf den Ausgangswert x, ohne einen zusätzlichen aufrufenden Knoten. Zu

```
1  input int in1, in2;
2  output int out, out2;
3
4  node compute(int a, b)
5               returns(int x, y) {
6    x = a + b;
7    y = a * b;
8  }
9  call = compute(in1, in2);
10 out = call.x;
11 out2 = call.y;
```

(a) Vorgestellte Notation (mit Hilfe von aufrufenden Knoten)

```
1  input int in1, in2;
2  output int out, out2;
3
4  node compute(int a, b)
5               returns(int x, y) {
6    x = a + b;
7    y = a * b;
8  }
9
10 out = compute(in1, in2).x;
11 out2 = compute(in1, in2).y;
```

(b) Mögliche vereinfachte Notation (ohne Zwischenschritt)

Abbildung 7.2. Beispiel zur Vereinfachung der Notation von Datenfluss in SCCharts

beachten wäre dann, dass der Aufruf des definierenden Knotens mit den gleichen Parametern nicht zwingend anschließend zwei mal visualisiert und transformiert werden muss. In der bestehenden Implementierung erfolgt die Visualisierung und Transformation über die aufrufenden Knoten. Gibt es wie in Zeile elf ersichtlich einen zweiten Zugriff auf den gleichen Knoten, sollte man dies bei der vereinfachten Notation mit beachten.

Eine weitere sich anschließende Vereinfachung ist die direkte Weiterverwendung von Ausgängen eines definierenden oder referenzierenden Knotens. In Abbildung 7.3 ist dafür ein Beispiel zu sehen, in dem der globale Ausgangswert out2 aus der Addition des Ausgangswerts eines Knotens und eines globalen Eingangswerts berechnet wird. Der Aufruf des Knotens compute entspricht dabei dem definierenden Knoten aus dem obigen Beispiel. Wie auf der rechten Seite der Abbildung zu sehen ist, wäre eine verkürzte Notation, ohne aufrufenden Knoten und die Belegung des Ausgangswertes out, kompakter und weiterhin lesbar. Eine Verwendung beider Möglichkeiten zur Modellierung durch das Metamodell und die Grammatik zuzulassen wäre ebenfalls denkbar.

```
1 call = compute(in1, in2);
2 out = call.x;
3 out2 = out + in1;
```

(a) Vorgestellte Notation (mit Zwischenschritten)

```
1 out2 = compute(in1, in2).x + in1;
```

(b) Mögliche vereinfachte Notation (ohne Zwischenschritte)

Abbildung 7.3. Beispiel zur Vereinfachung der Notation von Datenfluss in SCCharts

Ausdrücke beim Knotenaufruf als Parameter mit übergeben Eine weitere Möglichkeit die Modellierung von Datenfluss noch weiter zu vereinfachen wäre die Verwendung von Ausdrücken als Parameter bei Knotenaufrufen. In Abbildung 7.4 ist dies zur Verdeutlichung exemplarisch dargestellt. Auf der linke Seite sieht man die in dieser Arbeit vorgestellten Notation, in der nur Variablen als Parameter mit übergeben werden können. Auf der rechten Seite sieht man eine mögliche Vereinfachung des Knotenaufrufs `compute` (aus dem ersten Beispiel), bei dem die Berechnung von `in1 * in2` vorher nicht separat notiert werden muss. Die Vereinfachung des Knotenaufrufs ohne einen extra Knoten ist in diesem Beispiel ebenfalls mit enthalten.

```
1 t = in1 * in2;
2 call = compute(t, in3);
3 out = call.x;
```

(a) Vorgestellte Notation (mit Zwischenschritten)

```
1 out = compute(in1 * in2, in3).x + in1;
```

(b) Mögliche vereinfachte Notation (Ausdruck wird als Parameter mit übergeben)

Abbildung 7.4. Beispiel zur Erweiterung der Notation von Datenfluss in SCCharts

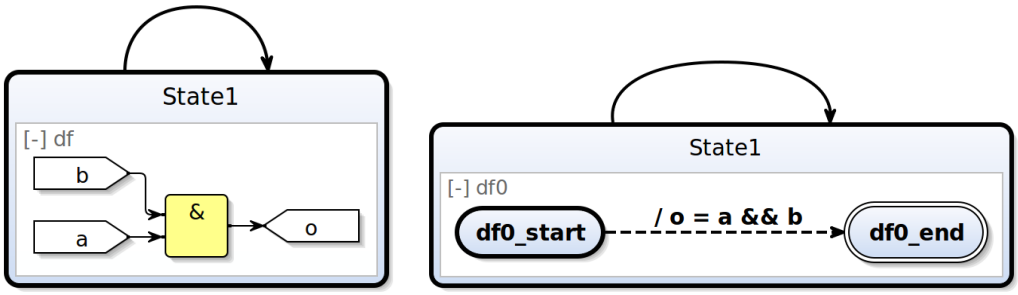
Erweiterte Funktionalität des ScopeProviders Der `ScopeProvider` ist eine hilfreiche Funktion zur Vereinfachung der Modellierung. In der bisherigen Implementierung, wie in Abschnitt 5.1.2 beschrieben, bietet er bereits an zwei Stellen dem Entwickler unterstützende Auswahllisten.

Ein weiterer Anwendungsfall für den `ScopeProvider` wäre bei einer Verallgemeinerung der Verwendung von definierenden und aufrufenden Knoten denkbar. In der bisherigen Implementierung kann ein aufrufender Knoten nur auf einen definierenden Knoten innerhalb der gleichen Datenflussumgebung verweisen. Erlaubt man den Zugriff auch auf definierenden Knoten in anderen Umgebungen oder auf erster Ebene im SCChart, dann wäre eine Auswahlliste aller möglichen Knoten, auf die verwiesen werden kann, sehr hilfreich.

7.2.2. Ausbau der Referenztransformation

Die Transformation von Datenfluss in bereits bestehende Konstrukte der SCCharts ist ein wichtiger Aspekt um schlussendlich sequentiellen C-Code zu generieren. Aufgrund der Komplexität der zu beachtenden Faktoren und Bedingungen ist diese Transformation in der vorliegenden Arbeit nur in einer grundlegenden Form implementiert.

Im Allgemeinen wird angenommen, dass der Datenfluss in einer synchronen Sprache instantan abläuft. Des Weiteren wird angenommen, dass diese Ausführung von Datenfluss in jedem Tick abläuft. Die Transformation von Datenfluss in klassische SCCharts berücksichtigt bereits die instantane Ausführung. Die generierten Transitionen, deren Effekte die Datenflussberechnungen darstellen, sind *immediate*, also instantan. Die Ausführung von Datenfluss in jedem Tick bleibt in der vorliegenden Implementierung weitestgehend dem Entwickler überlassen. In Abbildung 7.5a ist eine Datenflussumgebung in einem Zustand mit Selbsttransition zu sehen. Die hier beispielhaft verwendete Transition ist vom Typ *weak* abort. Das SCChart sieht nach der Referenztransformation wie in Abbildung 7.5b aus. Wie zu erkennen ist, würde die Datenflussgleichung instantan ausgeführt werden, wenn der entsprechende Zustand `State1` aktiv ist. Durch die Selbsttransition wird der Datenfluss dann auch in jedem Tick ausgeführt.



(a) Datenflussumgebung in einem Zustand mit Selbsttransition

(b) Datenflussumgebung in einem Zustand mit Selbsttransition nach der Referenztransformation

Abbildung 7.5. Beispiel zur Ausführung von Datenfluss in jedem Tick

Eine mögliche Erweiterung der Referenztransformation ist es, dem Entwickler die Modellierung der Selbsttransition abzunehmen. Diese Transition könnte bei der Transformation automatisch generiert werden. Dabei ist noch zu bedenken, von welchem Typ die Transition sein muss. Der Transitionstyp eines *weak abort* wurde hier im Beispiel verwendet. Weitere Transitionstypen sind der *strong abort* und die Terminierungstransition.

Für die Transformation von Datenflussgleichungen ließe sich auch folgende Erweiterung verwenden. Wie in Abschnitt 4.4.2 beschrieben, kann der Zielzustand der instantanen Transition eine normale, um einen Tick verzögerte, Transition

zurück zum Startzustand erhalten. Die Datenflussberechnungen würden dann ebenfalls in jedem Tick ausgeführt werden, ohne das eine Selbsttransition des umfassenden Zustandes nötig ist. Diese Erweiterung lässt sich auch auf definierende Knoten, die Datenfluss enthalten, anwenden.

Ein weiterer Punkt für ein effizienteres transformiertes SCChart ist die Minimierung von den generierten nebenläufigen Regionen. Bisher wird für jeden aufrufenden Knoten eine eigene neue Region erzeugt. Die Datenflussgleichungen der direkten Datenflussmodellierung werden bereits in einer einzelnen Region zusammengefasst. Zu überlegen ist, ob sich in einem ersten weiteren Schritt die generierten Regionen der aufrufenden Knoten ebenfalls in einer Region zusammenfassen lassen. In einem zweiten Schritt könnte man analysieren, ob und wie man die nebenläufigen Regionen von Knoten und Gleichungen sinnvoll zusammenfassen kann.

Durch die freie Wahl in der Verhaltensbeschreibung in einem definierenden Knoten mit Kontrollfluss ist folgende Problemstellung noch näher zu betrachten. Wenn angenommen wird, dass der Datenfluss in jedem Tick instantan durchläuft, wie muss sich die Transformation verhalten, wenn der modellierte Kontrollfluss verzögert abläuft. Es muss gewährleistet werden, dass alle Ausgänge des Knotens in jedem Tick einen gültigen Wert haben. Eine Möglichkeit um dies zu gewährleisten ist es, dass alle Ausgangswerte ihren alten Wert behalten, sollte das modellierte Verhalten nicht in einem Tick vollständig durchlaufen werden. Dabei ist vor allem zu beachten, dass alle Ausgangswerte einen voreingestellten Standardwert im ersten Tick bekommen, damit keine undefinierten Werte weitergegeben werden. Eine striktere Alternative wäre es, die Beschreibung von Kontrollfluss innerhalb definierender Knoten nur zu erlauben, wenn das Verhalten unverzögert ablaufen kann. Ein SCChart in dem dies nicht der Fall ist müsste als nicht valide gewertet werden, wenn diese Voraussetzung nicht eingehalten wird.

Danksagung

Abschließend möchte ich mich an dieser Stelle bei allen bedanken, die mich während meines Studiums und besonders bei der Anfertigung meiner Diplomarbeit unterstützt haben.

Ganz besonders gilt mein Dank Herrn Prof. Dr. Reinhard von Hanxleden, der mir ermöglicht hat meine Diplomarbeit am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme zu schreiben.
Ebenso danke ich meinen Betreuern Insa Fuhrmann und Steven Smyth für ihre Hilfestellungen, Antworten auf meine Fragen und das Korrekturlesen meiner Arbeit. Ich bedanke mich bei Brigitte Scheidemann und Priv.-Doz. Dr. Frank Huch, für ihre stete Unterstützung in allen organisatorischen Belangen während meiner Studienzeit.
Ein herzlicher Dank geht an Lilly, Sebastian und Jan-Christoph, für Kost und Logis und die vielen Stunden die wir gemeinsam verbringen konnten. Dank gebührt ebenso meinen Eltern, für ihre immerwährende Unterstützung, ganz besonders in den letzten zwei Jahren meiner Studienzeit.
Nicht zuletzt geht mein ganz besonderer Dank an Sandra, Leon und Luca. Für ihre unablässige Treue und den Glauben an mich und meine Arbeit.

Literaturverzeichnis

[And96a] André, C.: *Representation and Analysis of Reactive Behaviors: A Synchronous Approach*. In: *Computational Engineering in Systems Applications (CESA)*, Seiten 19–29, Lille (F), July 1996. IEEE-SMC.

[And96b] André, Charles: *SyncCharts: a Visual Representation of Reactive Behaviors*. Technischer Bericht RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[BB91] Benveniste, Albert und Gerard Berry: *The synchronous approach to reactive and real-time systems*. In: *Proceedings of the IEEE*, Seiten 1270–1282, 1991.

[BCE+03] Benveniste, Albert, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, Robert und De Simone: *The synchronous languages 12 years later*. In: *Proceedings of The IEEE*, Seiten 64–83, 2003.

[BCL+05] Brooks, Christopher, Adam Cataldo, Edward A. Lee, J. Liu, Xiaojun Liu, Stephen Neuendorffer und Haiyang Zheng: *HyVisual: A Hybrid System Visual Modeler*. Technischer Bericht UCB/ERL M05/24, EECS Department, University of California, Berkeley, Jul 2005. http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/9574.html.

[BdS91] Boussinot, F. und R. de Simone: *The ESTEREL language*. Proceedings of the IEEE, 79(9):1293–1304, Sep 1991, ISSN 0018-9219.

[Ber07] Berry, Gérard: *SCADE: Synchronous Design and Validation of Embedded Control Software*. In: Ramesh, S. und Prahladavaradan Sampath (Herausgeber): *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, Seiten 19–33. Springer Netherlands, 2007, ISBN 978-1-4020-6253-7.

[BG92] Berry, Gérard und Georges Gonthier: *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*. Sci. Comput. Program., 19(2):87–152, November 1992, ISSN 0167-6423.

[BK02] Brandes, Ulrik und Boris Köpf: *Fast and Simple Horizontal Coordinate Assignment*. In: Mutzel, Petra, Michael Jünger und Sebastian Leipert (Herausgeber): *Graph Drawing*, Band 2265 der Reihe *Lecture Notes in Computer Science*, Seiten 31–44. Springer Berlin Heidelberg, 2002, ISBN 978-3-540-43309-5. http://dx.doi.org/10.1007/3-540-45848-4_3.

[BP88] Bergerand, J. L. und E. Pilaud: *Saga: a software development environment for dependability in automatic control*. In: *Proc. Safecomp'88*. Pergamon Press, 1988.

[CP03] Colaço, Jean Louis und Marc Pouzet: *Clocks as First Class Abstract Types*. In: Alur, Rajeev und Insup Lee (Herausgeber): *Embedded Software*, Band 2855 der Reihe *Lecture Notes in Computer Science*, Seiten 134–155. Springer Berlin Heidelberg, 2003, ISBN 978-3-540-20223-3. http://dx.doi.org/10.1007/978-3-540-45212-6_10.

[CPHP87] Caspi, P., D. Pilaud, N. Halbwachs und J. A. Plaice: *LUSTRE: A Declarative Language for Real-time Programming*. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, Seiten 178–188, New York, NY, USA, 1987. ACM, ISBN 0-89791-215-2. http://doi.acm.org/10.1145/41625.41641.

[CPP05] Colaço, Jean Louis, Bruno Pagano und Marc Pouzet: *A Conservative Extension of Synchronous Data-flow with State Machines*. In: *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, Seiten 173–182, New York, NY, USA, 2005. ACM, ISBN 1-59593-091-4. http://doi.acm.org/10.1145/1086228.1086261.

[EJL+03] Eker, J., J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs und Yuhong Xiong: *Taming heterogeneity - the Ptolemy approach*. Proceedings of the IEEE, 91(1):127–144, Jan 2003, ISSN 0018-9219.

[FvH10a] Fuhrmann, Hauke und Reinhard von Hanxleden: *On the Pragmatics of Model-Based Design*. In: *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, Band 6028 der Reihe *LNCS*, Seiten 116–140, 2010.

[FvH10b] Fuhrmann, Hauke und Reinhard von Hanxleden: *Taming Graphical Modeling*. In: *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, Band 6394 der Reihe *LNCS*, Seiten 196–210. Springer, October 2010.

[Gam10] Gamatié, Abdoulaye: *Designing Embedded Systems with the SIGNAL Programming Language - Synchronous, Reactive Specification.* Springer, 2010, ISBN 978-1-4419-0940-4.

[GGBM91] Guernic, Paul Le, Thierry Gautier, Michel Le Borgne und Claude Le Maire: *Programming Real-Time Applications with SIGNAL*. In: *Proceedings of the IEEE*, Seiten 1321–1336, 1991.

[Har87] Harel, David: *Statecharts: A Visual Formalism for Complex Systems*. Sci. Comput. Program., 8(3):231–274, Juni 1987, ISSN 0167-6423. http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[HCRP91] Halbwachs, N., P. Caspi, P. Raymond und D. Pilaud: *The synchronous dataflow programming language LUSTRE*. In: *Proceedings of the IEEE*, Seiten 1305–1320, 1991.

[Lee09] Lee, Edward A.: *Finite State Machines and Modal Models in Ptolemy II*. Technischer Bericht UCB/EECS-2009-151, UC Berkeley, November 2009.

[LZ05] Lee, EdwardA. und Haiyang Zheng: *Operational Semantics of Hybrid Systems*. In: Morari, Manfred und Lothar Thiele (Herausgeber): *Hybrid Systems: Computation and Control*, Band 3414 der Reihe *Lecture Notes in Computer Science*, Seiten 25–53. Springer Berlin Heidelberg, 2005, ISBN 978-3-540-25108-8. http://dx.doi.org/10.1007/978-3-540-31954-2_2.

[Mot09] Motika, Christian: *Semantics and execution of domain specific models–KlePto and an execution framework*. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik, 2009.

[MR98] Maraninchi, Florence und Yann Rémond: *Mode-Automata: About Modes and States for Reactive Systems*. In: *European Symposium On Programming (ESOP)*, Lisbon (Portugal), mar 1998. Springer Verlag, LNCS 1381.

[MR03] Maraninchi, F. und Y. Rémond: *Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems*. Science of Computer Programming, (46):219–254, 2003.

[MSvH14] Motika, Christian, Steven Smyth und Reinhard von Hanxleden: *Compiling SCCharts—A Case-Study on Interactive Model-Based Compilation*. In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, Band 8802 der Reihe *LNCS*, Seiten 443–462, Corfu, Greece, October 2014.

[MvHH13] Motika, Christian, Reinhard von Hanxleden und Mirko Heinold: *Programming Deterministice Reactive Systems with Synchronous Java (Invited Paper)*. In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*, IEEE Proceedings, Paderborn, Germany, 17/18 June 2013.

[PBEB07] Potop-Butucaru, Dumitru, Stephen A. Edwards und Gerard Berry: *Compiling Esterel*. Springer Publishing Company, Incorporated, 1. Auflage, 2007, ISBN 0387706267, 9780387706269.

[Sch11] Schulze, Christoph Daniel: *Optimizing automatic layout for data flow diagrams*. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik, 2011.

[Spö09] Spönemann, Miro: *On the automatic layout of data flow diagrams*. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik, 2009.

[SSvH12] Schneider, Christian, Miro Spönemann und Reinhard von Hanxleden: *Transient View Generation in Eclipse*. In: *Proceedings of the First Workshop on Academics Modeling with Eclipse*, Kgs. Lyngby, Denmark, July 2012.

[SSvH13] Schneider, Christian, Miro Spönemann und Reinhard von Hanxleden: *Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice*. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*, San Jose, CA, USA, 15–19 9 2013.

[vHDM+13] Hanxleden, Reinhard von, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer und Owen O'Brien: *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications*. Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013. ISSN 2192-6247.

[vHDM+14] Hanxleden, Reinhard von, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer und Owen O'Brien: *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications*. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.

[vHMA+13] Hanxleden, Reinhard von, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer und Owen O'Brien: *Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation*. In: *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, Seiten 581–586, Grenoble, France, March 2013. IEEE.